.NET FULL STACK

Development Program

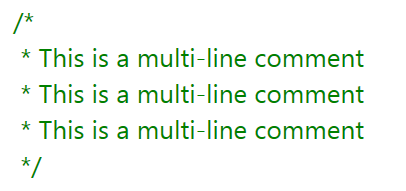
Day 2 C# Basic

Outline

* Comment
* Variable and Data Type
* Operator
* Flow Control
* Keywords

# Comment

**In C#, there are 3 types of comments:**

● 

Single-line Comments ( // )

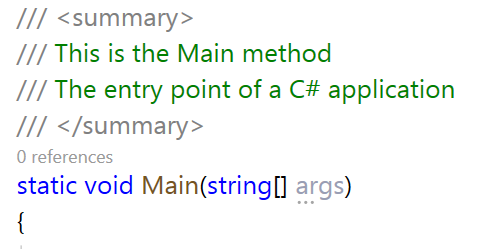
Single-line comments start with a double slash //. The compiler ignores everything after // to the end of the line.

●

Multi-line Comments ( /\* \*/ )

Multi-line comments start with /\* and ends with \*/. Multi-line comments

can span over multiple lines.

●

XML Comments ( /// )

XML documentation comment is a special feature in C#. It starts with a triple slash /// and is used to categorically describe a piece of code.. This is done using XML tags within a comment. These comments are then, used to create a separate XML documentation file.

# Variable and Data Type

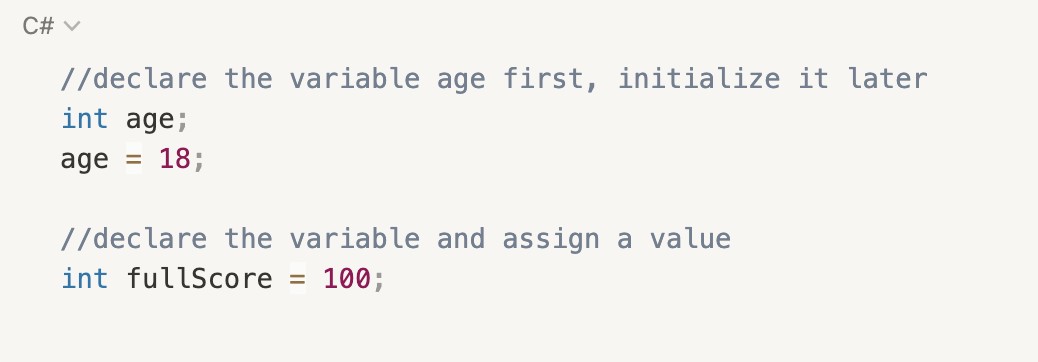
Variable

* A variable is a named memory location that we can create in order to store

data.

* Every variable has a type that determines what values can be stored in the

variables.

●

[data type] [variable name];

Declaration

Initialization

●

[variable name] = [value];

●

[data type] [variable name] = [value];

Combined

Variables and Data Types

●

Types of variables

* + A data type specifies the type of data that a variable can store.

○

Two kinds of types:

■

*simple types*, *enum types*, *struct types*, *nullable value types*, and *tuple value types.*

* *Reference types:*

*string, class types*, *interface types*, *array types*, and *delegate types.*

Value types:

# Value Type

* Simple Types

[○](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/integral-numeric-types)

[Signed integral](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/integral-numeric-types): sbyte, short, int, long

[○](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/integral-numeric-types)

[Unsigned integral](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/integral-numeric-types): byte, ushort, uint, ulong

[○](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types)

[floating-point](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types): float, double

[○](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types)

[High-precision decimal floating-point](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types): decimal

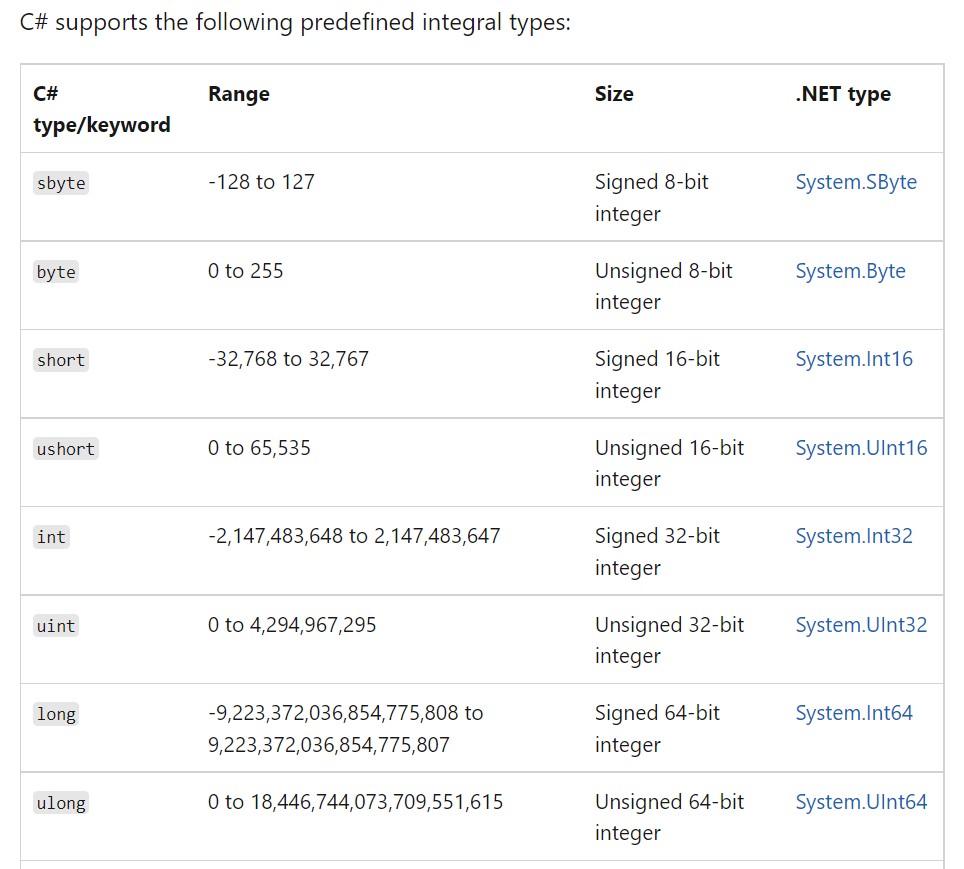
[○](https://docs.microsoft.com/en-us/dotnet/standard/base-types/character-encoding-introduction)

[Unicode characters](https://docs.microsoft.com/en-us/dotnet/standard/base-types/character-encoding-introduction): char

○

Boolean: bool

# Signed and Unsigned Integral

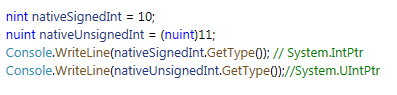


nint & nuint

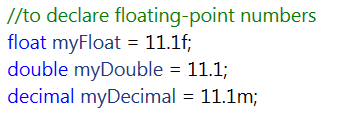
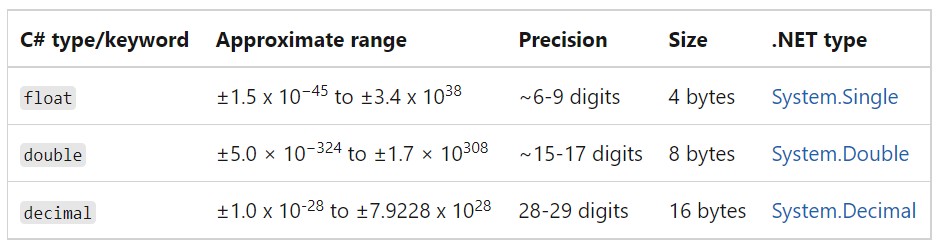
* Starting in C# 9.0, you can use the nint and nuint keywords to define ***native-sized integers***. These are 32-bit integers when running in a 32-bit process, or 64-bit integers when running in a 64-bit process.
* The native-sized integer types are represented internally as the .NET types System.IntPtr and

System.UIntPtr.

* Range:
  + For **nint**: Int32.MinValue to Int32.MaxValue.
  + For **nuint**: UInt32.MinValue to UInt32.MaxValue.
* There’s no direct syntax for native-sized integer literals. There’s no suffix to indicate that a literal is a native-sized integer, such as L/l to indicate a long. You can use implicit or explicit casts of other integer values instead.

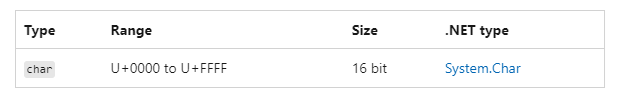


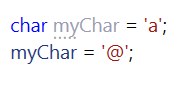
Floating-point numeric types

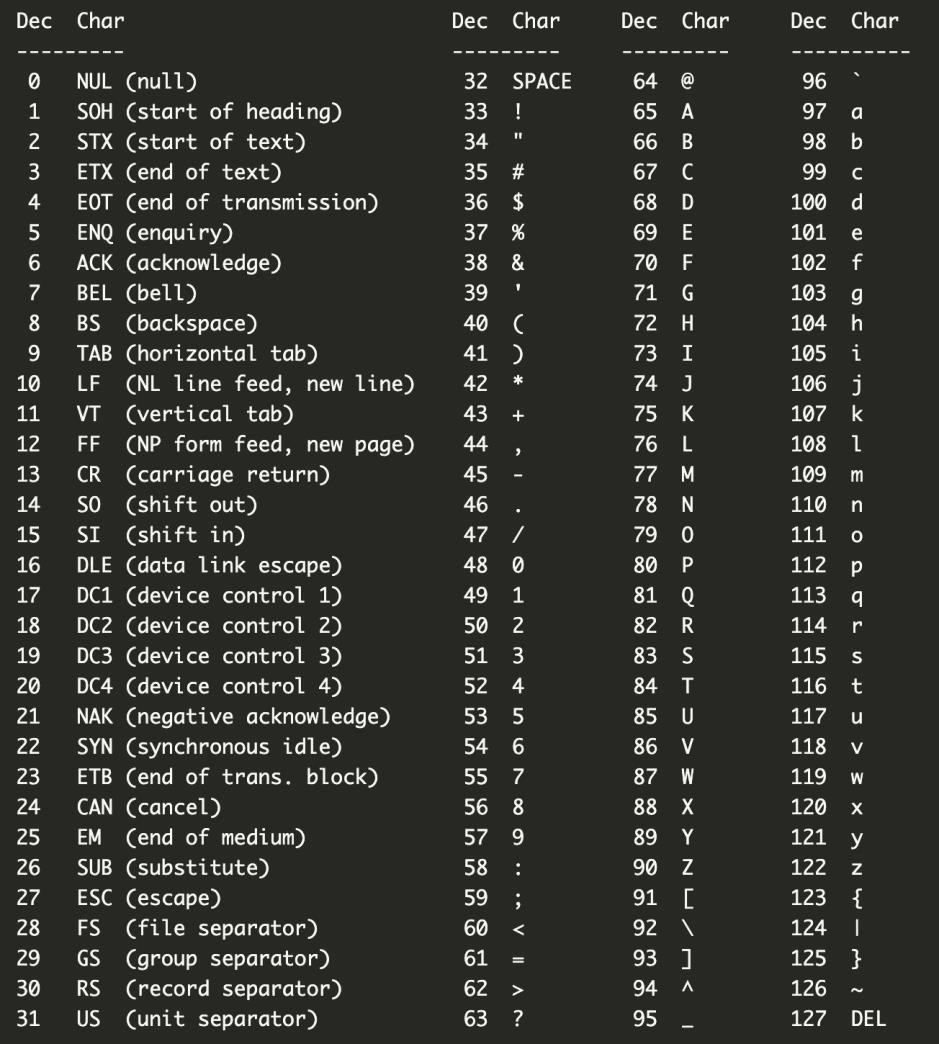


# Unicode Character: char

The **char** type keyword represents a Unicode UTF-16 character.







Type Conversion and Casting

* **Implicit conversions**: No special syntax is required because the conversion always succeeds, and “no data will be lost”. Examples include conversions from smaller to larger integral types.



* [**Explicit conversions (casting)**: Explicit conversions require a cast expression. Casting is required when information might be lost in t](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/type-testing-and-cast#cast-expression)he conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range.

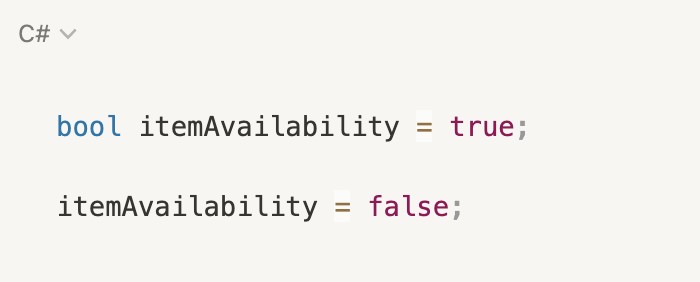


# Type Conversion and Casting

Boolean

The **bool** type keyword represents a Boolean value, which can be either true or false.

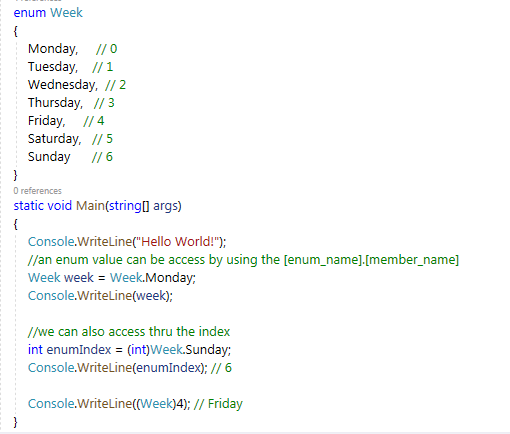




# Value Types

* [Value types](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-types)
  + [Simple types](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-types#built-in-value-types)
    - [Signed integral](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/integral-numeric-types): sbyte, short, int, long
    - [Unsigned integral](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/integral-numeric-types): byte, ushort, uint, ulong
    - [floating-point](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types): float, double
    - [High-precision decimal floating-point](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types): decimal
    - [Unicode characters](https://docs.microsoft.com/en-us/dotnet/standard/base-types/character-encoding-introduction): char
    - Boolean: bool
  + [Enum types](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum)
    - User-defined types of the form enum E {...}. An enum type is a distinct type with named constants. Every enum type has an underlying type, which must be one of the eight integral types. The set of values of an enum type is the same as the set of values of the underlying type.
  + [Struct types](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct)
    - User-defined types of the form struct S {...}
  + [Nullable value types](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/nullable-value-types)
    - Extensions of all other value types with a null value
  + [Tuple value types](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-tuples)
    - User-defined types of the form (T1, T2, ...)

# Enumeration type



An ***enumeration type*** (or *enum type*) is a [value type](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-types). To define an enumeration type, use

the ***enum*** keyword and specify the names of *enum members*

Structure type

* A structure type (or struct type) is a [value type](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-types) that can encapsulate data and related functionality. You use the struct keyword to

define a structure type.

●

and

We can have variables, methods,

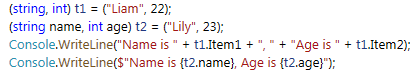
constructors defined in the struct.

●

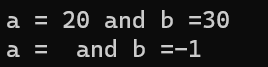
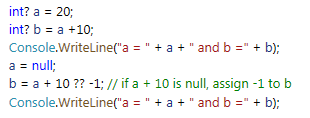
Struct doesn’t allow no-arg constructors.

# Tuple type

* + Available in C# 7.0 and later, tuple types are value types, the tuples feature provides concise syntax to group multiple data elements in a lightweight data structure.



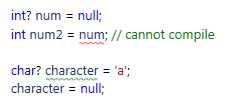
# Nullable Value Type - ?



* Starting from C# 8.0, A nullable value type represents all values of its underlying [value type](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-types)

and an additional [null](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/null) value. For example, you can assign any of the following three values to a

***bool?*** variable: **true**, **false**, or **null**.



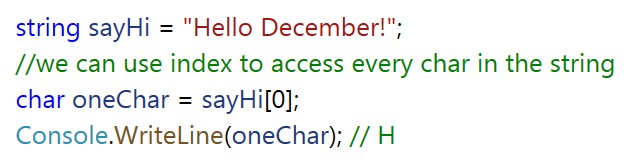
* The null-coalescing operator ?? returns the value of its left-hand operand if it isn’t null; otherwise, it evaluates the right-hand operand and returns its result. The ?? Operator doesn’t evaluate its right-hand operand if the left-hand operand evaluates to non-null.

# Reference Types

* + C# provides the following built-in reference types:
    - string
    - dynamic
    - object
  + The following keywords are used to declare reference types:
    - class
    - delegate
    - interface
    - record

# String type

The **string** type represents a sequence of zero or more Unicode characters (char).



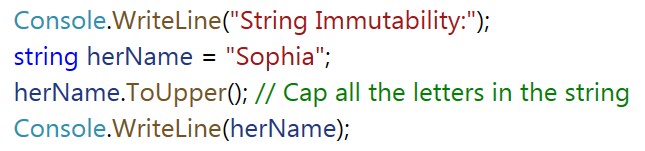
When

string.

Immutability of Strings

String objects are ***immutable***: they can't be changed after they've been created.

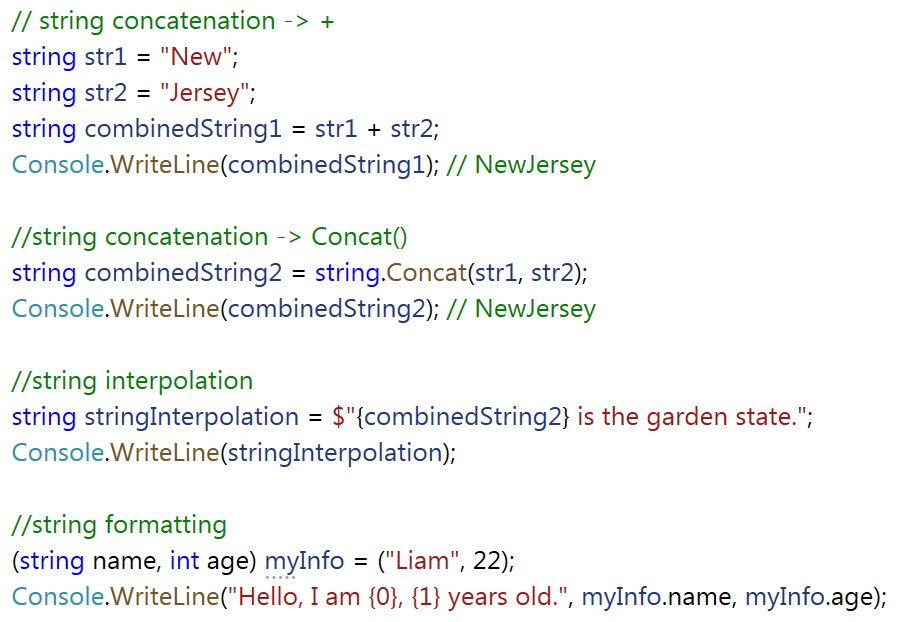
we assign a value to a string or modify the string, C# will create a new



String Interning

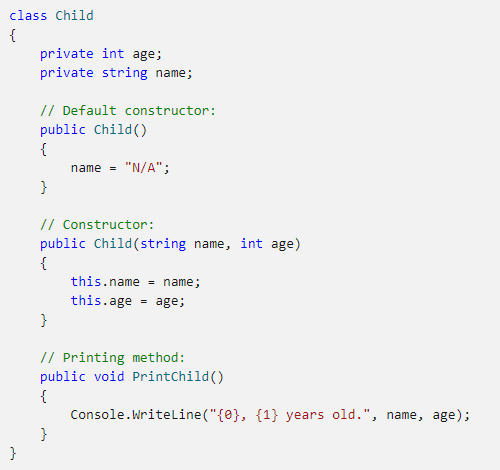
* + The common language runtime(CLR) conserves string storage by maintaining a table, called the ***intern pool***, that contains a single reference to each unique literal string declared or created programmatically in your program. Consequently, an instance of a literal string with a particular value only exists once in the system.
  + For example, if you assign the same literal string to several variables, the runtime retrieves the same reference to the literal string from the intern pool and assigns it to each variable.

# String Concatenation



* string.Format();
* string.Join();
* Append() method in StringBuilder;

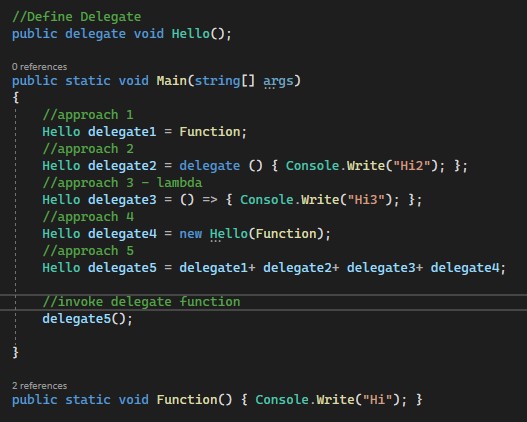
# Class

A class is a container that contains the block of code that includes field, method, constructor, etc.

# Delegate

A **delegate** is a reference type that can be used to encapsulate a named or an anonymous method.

The delegate must be instantiated with a method or lambda expression that has a compatible



return type and input parameters.

# Dynamic type

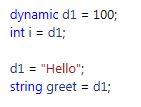
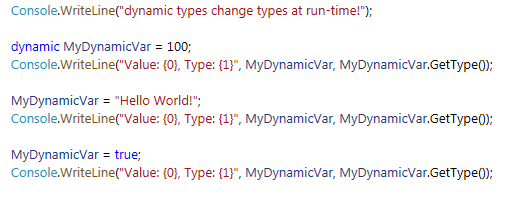
The **dynamic** type indicates that the use of the variable and references to its members

**escapes compile-time type checking.** Instead, it resolves the type at run time.

The **dynamic** types change types at run-time based on the assigned value, so **the type**

**dynamic only exists at compile-time, not run-time**.

The dynamic type variables are converted to other types implicitly.



Other Data Types

# Implicit type - Var

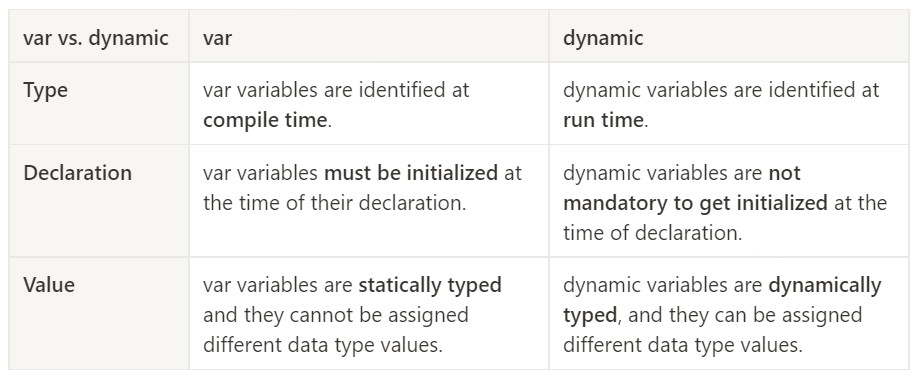
Beginning with C# 3, variables that are declared at method scope can have an implicit "type"

**var**. An implicitly typed local variable is strongly typed just as if you had declared the type

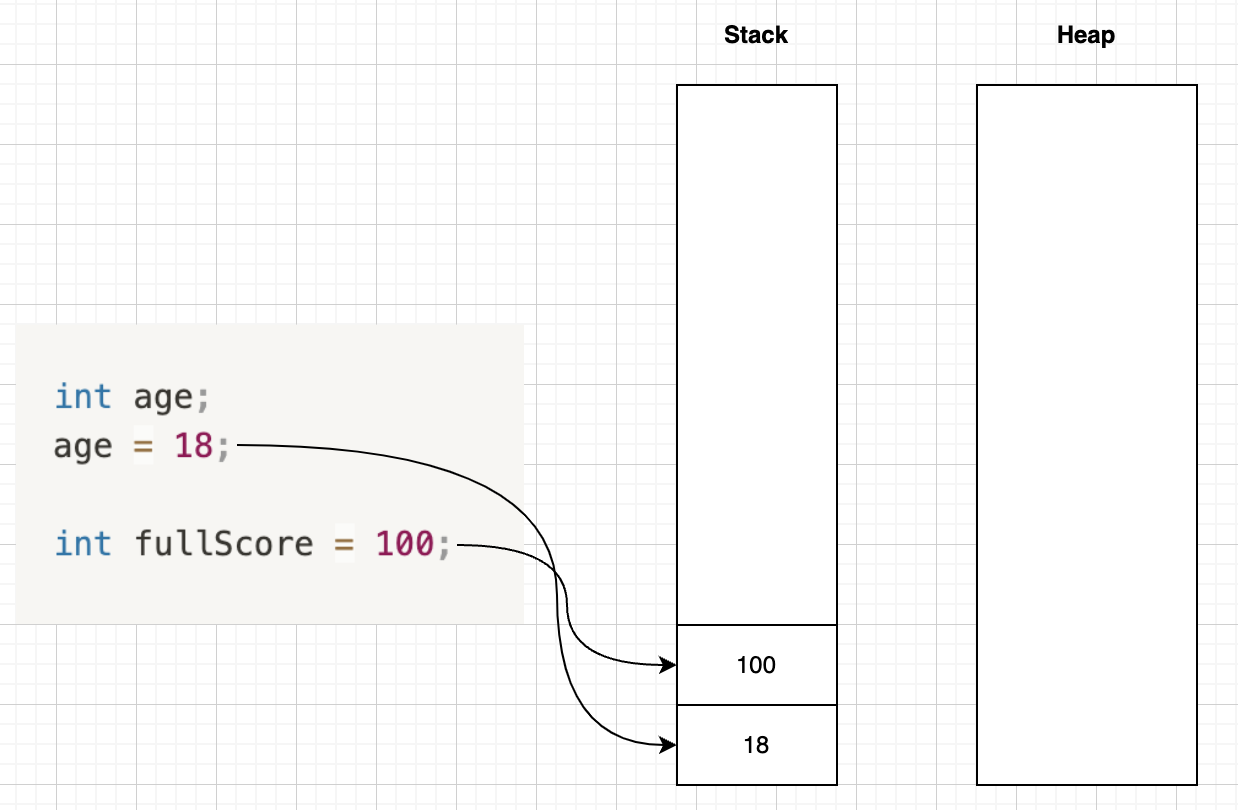
yourself, but the compiler determines the type.



Var vs. Dynamic



# Stack

* + Used for static memory

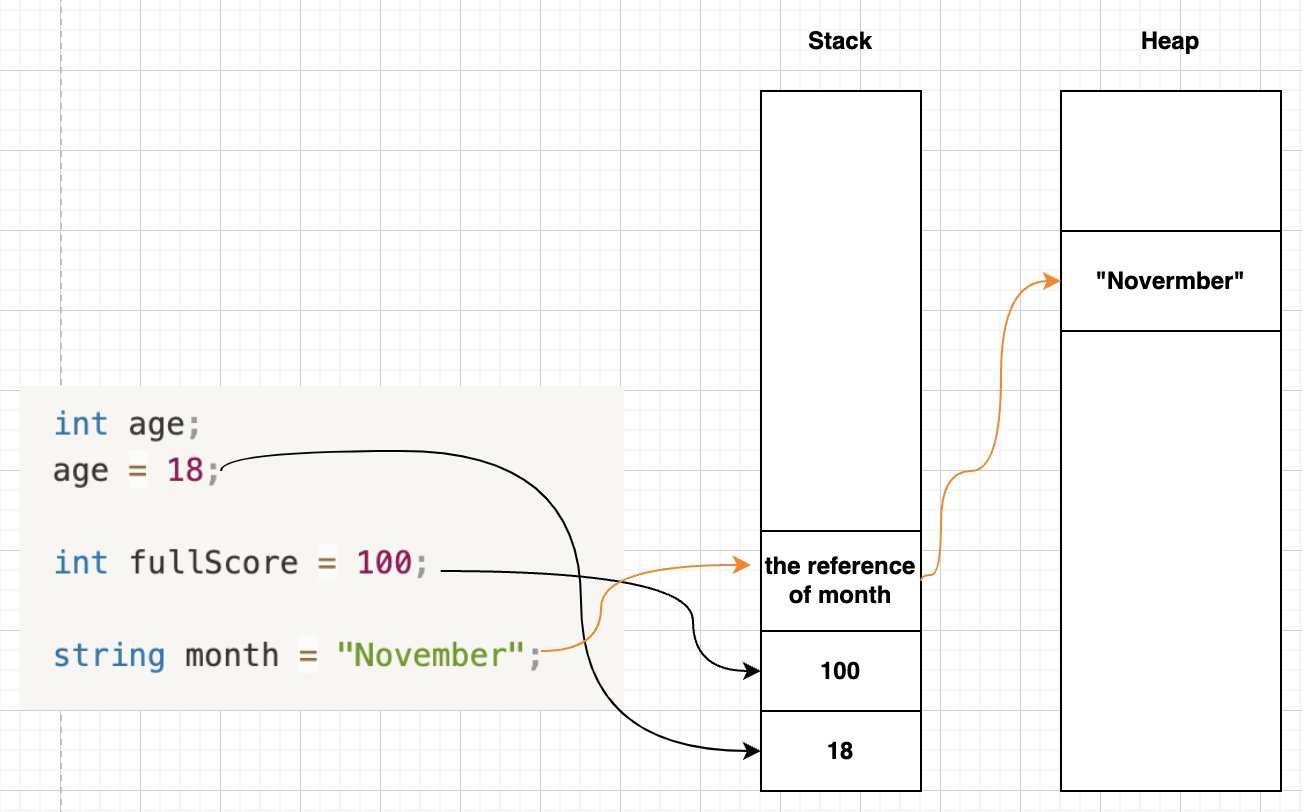
allocation

* + Contains data for value types
  + Contains references to

reference types(objects)

* + Relatively small
  + Access speed is fast

# Heap

* + Used for dynamic memory allocation
  + Contains the data of

reference types

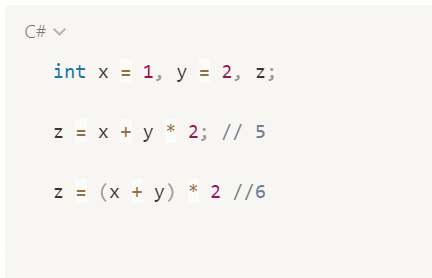
* + Relatively large
  + Access speed is slow

# Operator

Operator

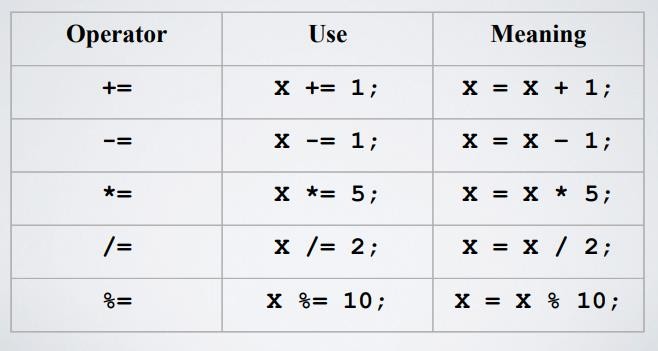
Arithmetic operations in C#

* Precedence: (\*, /, %) > (+, -)
* Parentheses: evaluate the innermost parenthesized expression first, and work

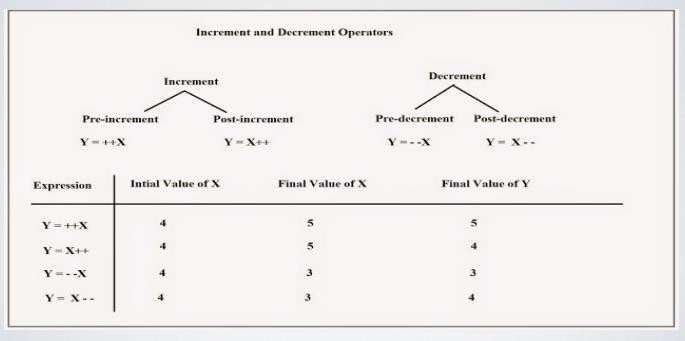
your way out through the levels of nesting

* No {} or [ ] in parentheses in C#

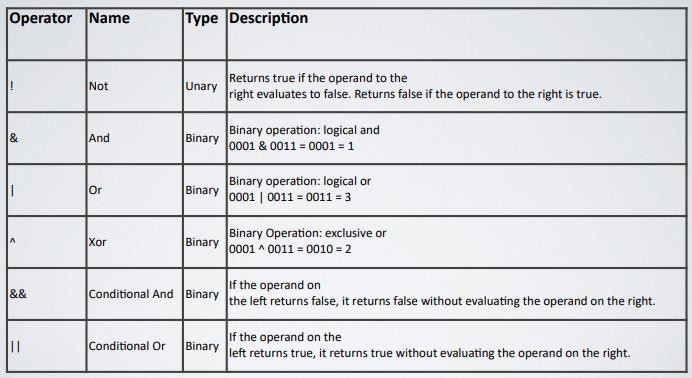
# Compound Arithmetic/ Assignment Operators



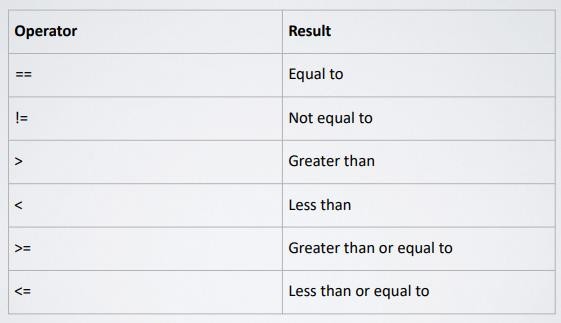
Increment and Decrement Operators



# Logical Operator



Relational Operators



# Flow Control

## Selection Statements

* + - if, switch

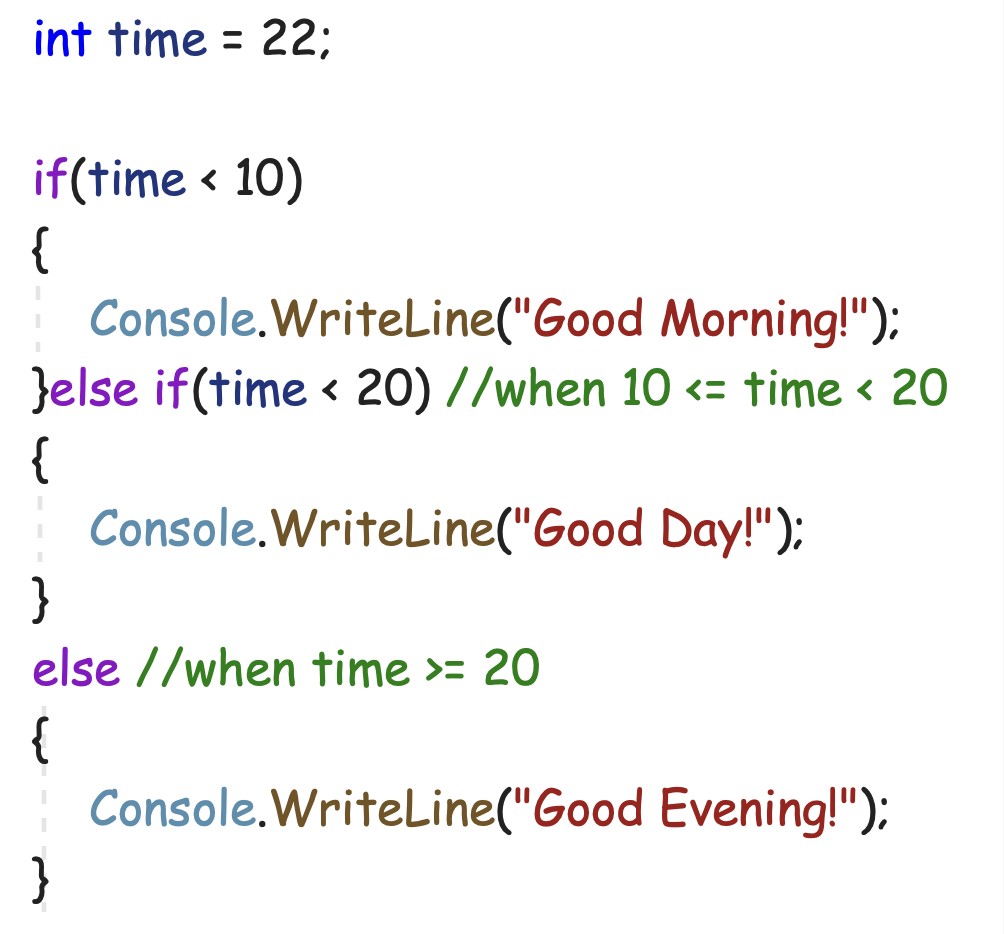
## Iteration Statements

* + - for, foreach, do-while, while

## Jump Statements

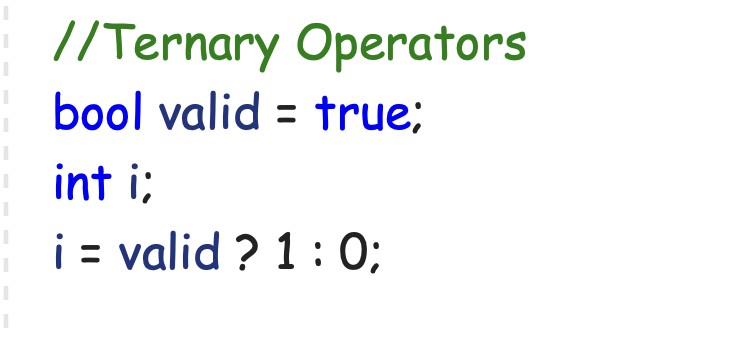
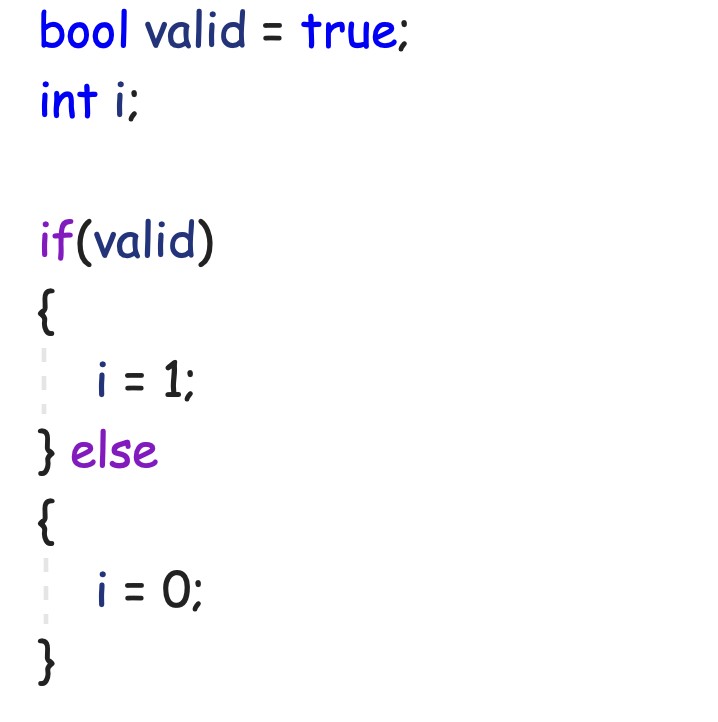
* + - break, continue, return, goto

# If

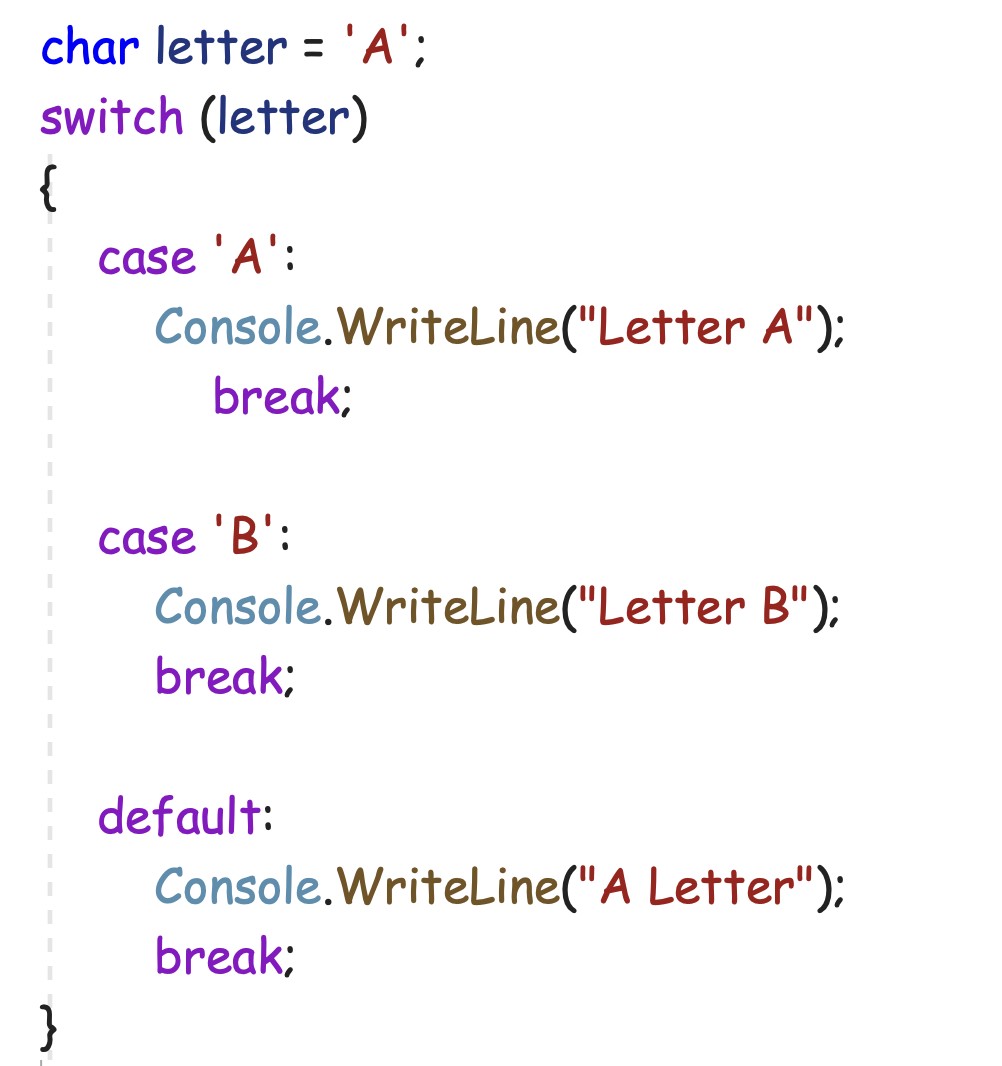


Ternary Operators

variable = condition ? expressionTrue : expressionFalse;



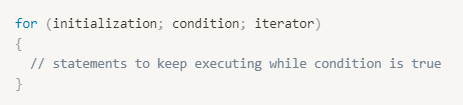
# Switch

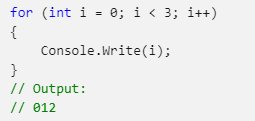


Iteration Statement

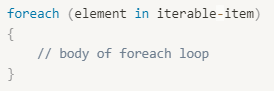
* + For
  + Foreach
  + While
  + Do-while

# For

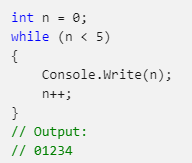
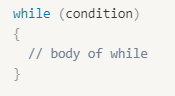




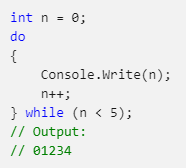
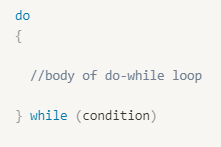
Foreach



# While



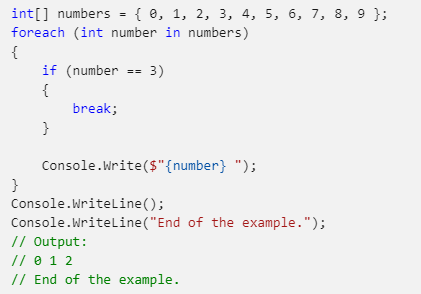
Do-while



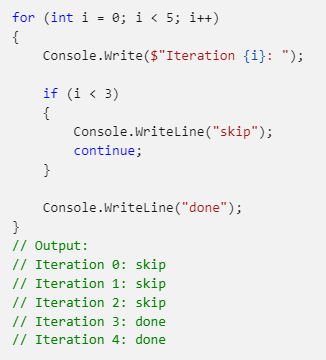
# Jump Statement

* + Break
  + Continue
  + Return
  + Goto

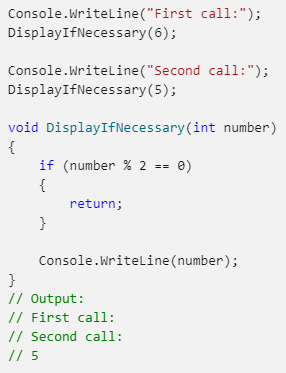
# Break



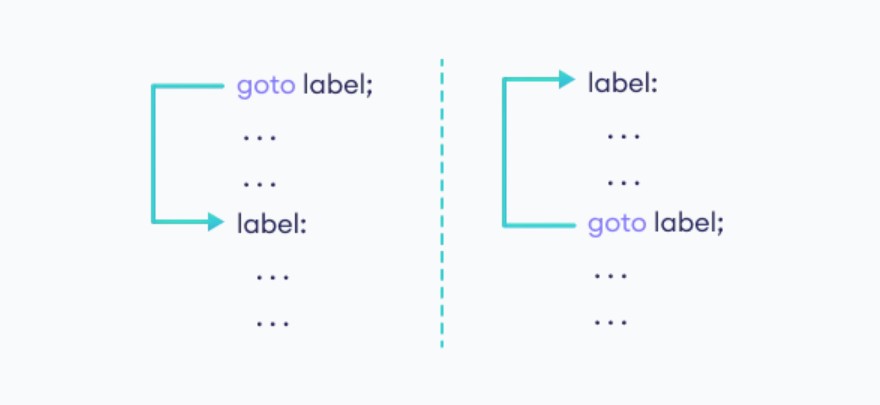
Continue



# Return



Goto



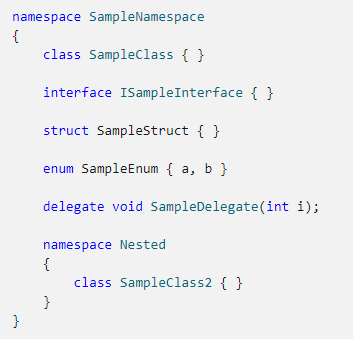
# Keywords

* Namespace
* Assembly
* Access Modifiers

# Namespace

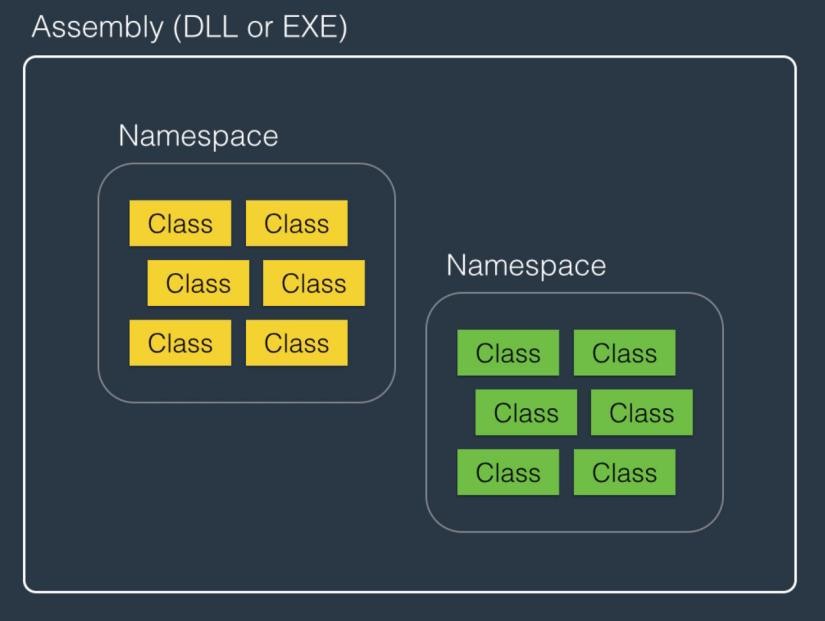
The **namespace** keyword is used to declare a scope that contains a set of related objects.

You can use a namespace to organize code elements and to create globally unique types.

* + You can think is as a package that includes classes that can be used elsewhere
  + \*namespace is often the file

name

# Assembly



An assembly is **a collection of types and resources that are built to work together and form a logical unit of functionality**. Assemblies take the form of executable (.exe) or dynamic link library (. dll) files and they are a single unit of deployment of .NET applications.

# Access Modifier

Access modifiers are keywords used to specify the declared accessibility of a

field, method, constructor & class

Those access modifier allows you to grant and prevent access:

●

public protected internal private

protected-internal privare-protected

●

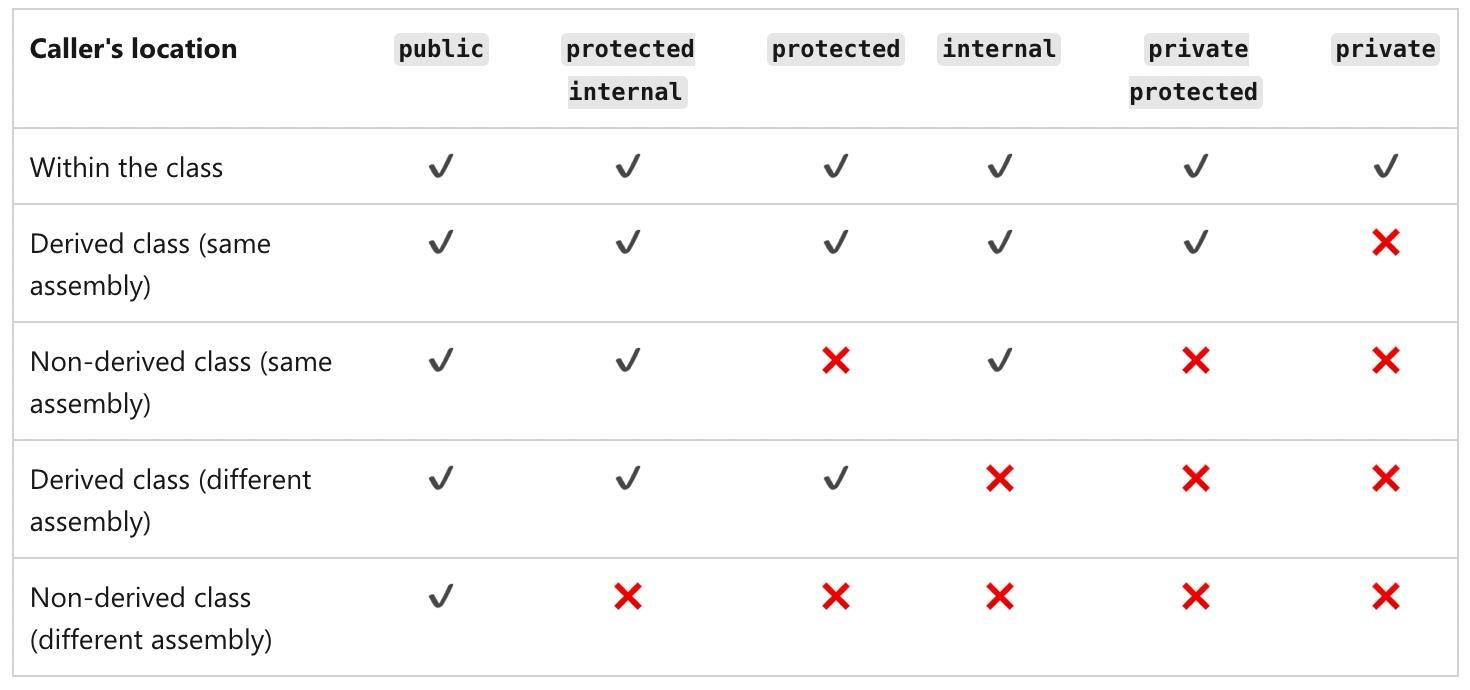
●

●

●

●

# Access Modifier



<https://code-maze.com/csharp-access-modifiers/>

# Recap

* Comment
* Variable and Data Type
  + int, long, float, double, enum, nullable, string
* Operator
* Flow Control
  + for, foreach, while
* Keywords
  + namespace, access modifier

Question?

.NET Full Stack

Development Program

Day 3 OOP

Outline

* Class and Object
* Object-oriented programming (OOP)
* Encapsulation
* Inheritance
* Polymorphism
* Abstract
* Object class

# OOP Intro

* OOP stands for Object-Oriented Programming
* The goal of OOP is to group up some data and operations as a single unit called an “Object”
* Object is a small unit in the program that represents a real-world entity

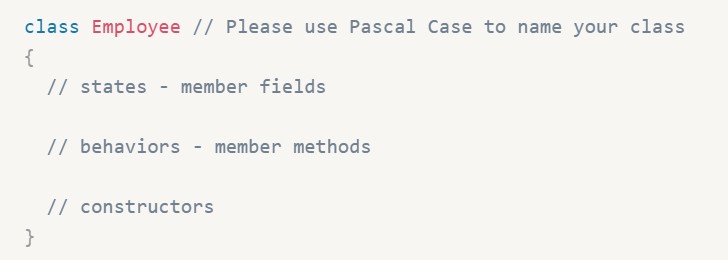
# Class

* C# is an object-oriented programming language, everything in C# is associated with classes and objects.
* A class is a “blueprint” for creating objects.

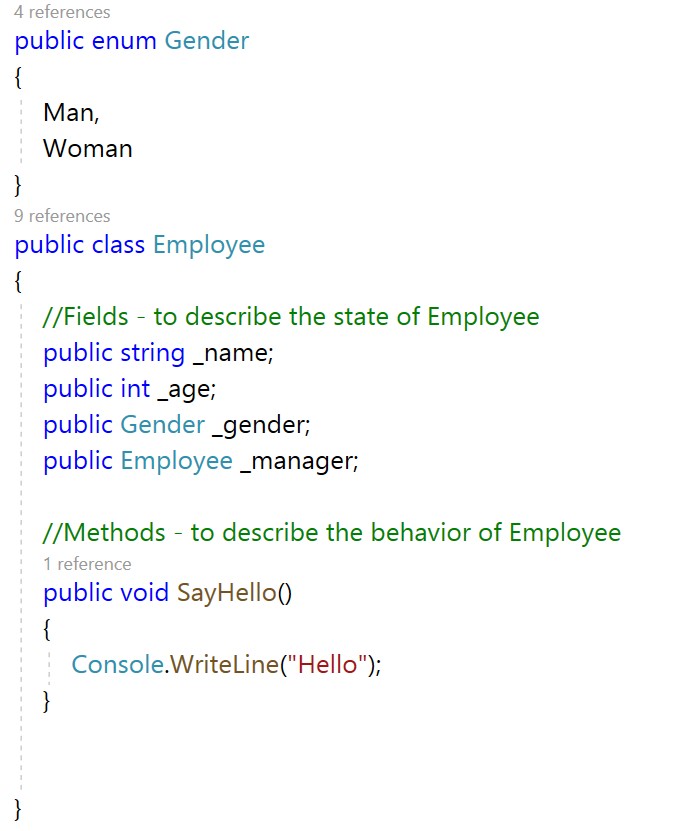


# Class

* To create a class we need to use the keyword – ***class***
* Normally we will create a class within a namespace



# Class Members



* Fields and methods inside classes are often referred to as “Class Members”.
* The variables inside a class are

called fields.

* Methods are used to perform certain actions.

# Constructors

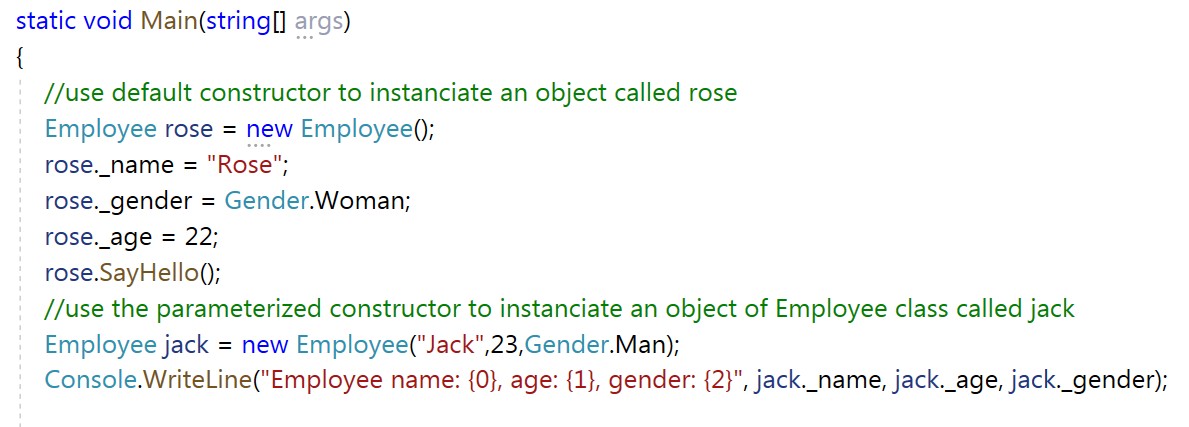


* A constructor is

a **special method** that is used to initialize objects.

* It can be used to set initial values for fields

# Object



* An **object** is the instance of the class, it has states and behaviors as well.
* The state of an object is stored in fields (variables), while methods (functions) display the

object's behavior

* To create an object of a class, specify the class name. followed by the object name, and

use the keyword new.

# Object vs Class



Static

* + declare a static member, which belongs to the **class** itself rather than to a specific object
* Static
  + Class
  + Compile time
* Non-static

■

Object

Runtime

■

# Static Variable

●

Static variable

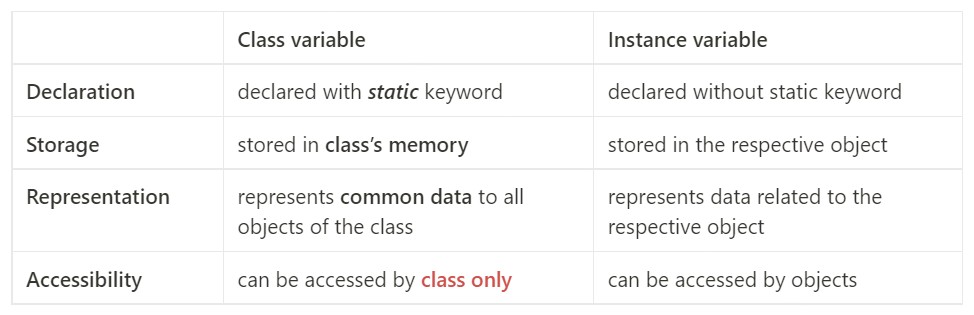
○

Static fields are stored outside of the object

○

Static fields are common to all objects of a class

# Class variable vs. Instance variable



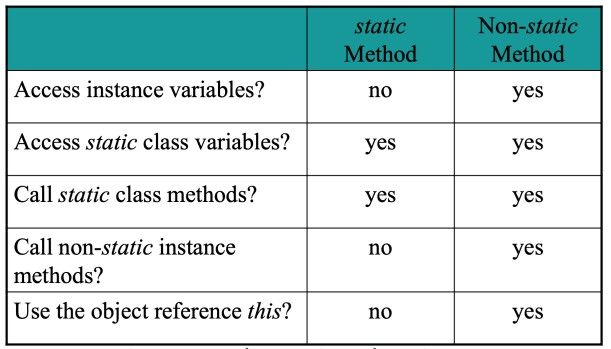
Static Method

●

Static method

○

static method vs. instance method



# Static Class

●

Static class

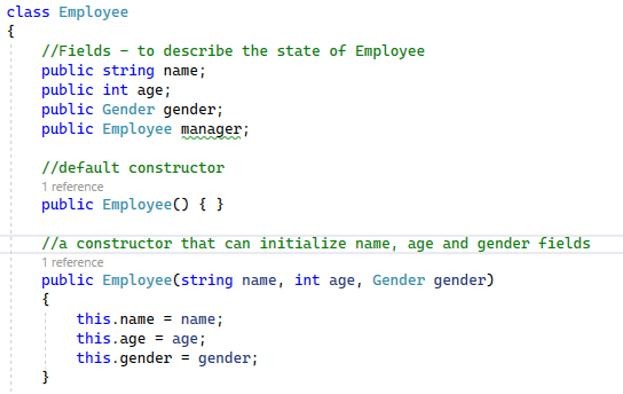
○

of the class must be static

If the static keyword is applied to a class, all the members

# This keyword

This keyword refers to the current object;

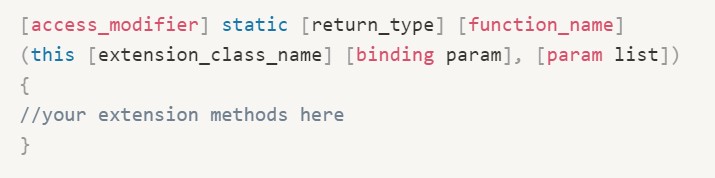


Usage:

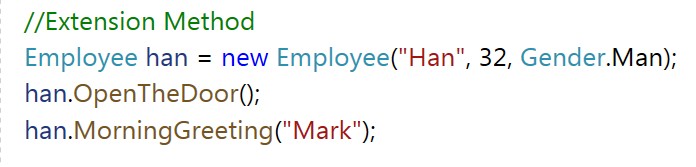
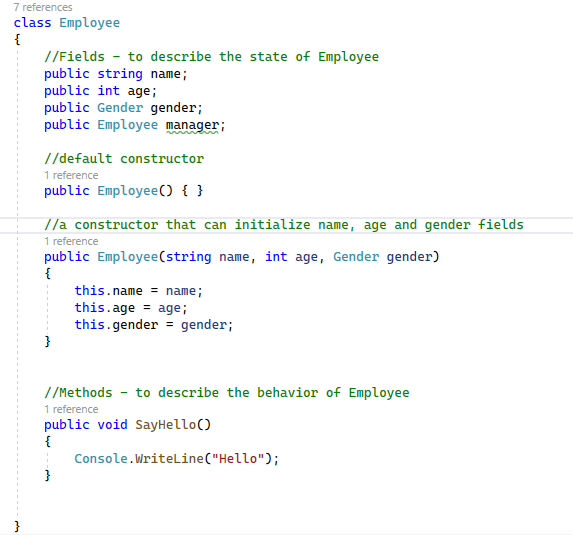
* ***this*** is often used to differentiate between the constructor parameters and class fields if they both have the same name.
* ***this*** is also used as a modifier of the first parameter of an extension method.

# Extension Method

* + It is a new feature that has been added in C# 3.0 which allows us to add new methods into a non-static class without editing the code of the class.
  + Extension methods must be defined only under the static class.
  + Syntax:



# Extension Method



Object-Oriented Programming

# Encapsulation

* + - Abstraction

# Inheritance

* + - Polymorphism

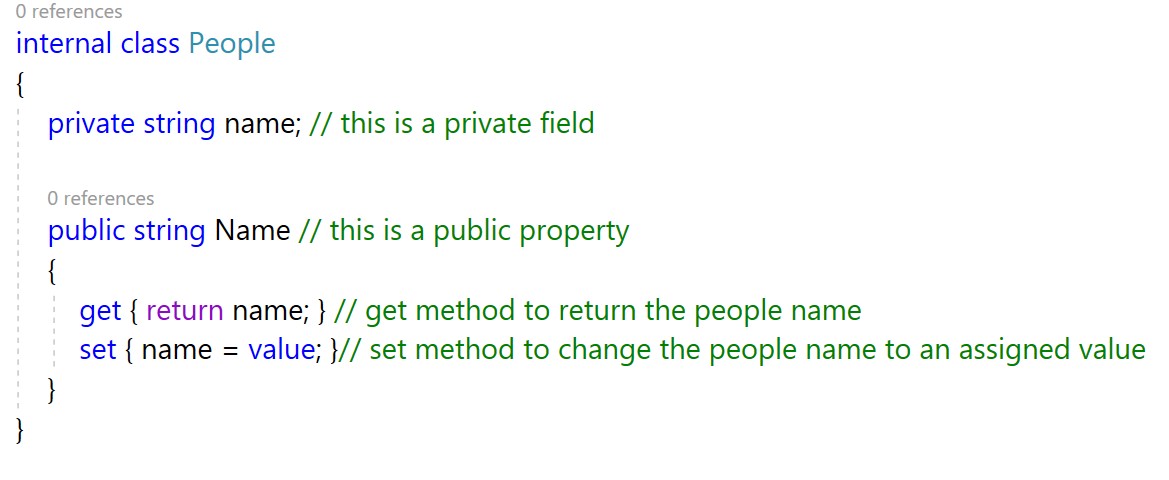
# Encapsulation

* + Encapsulation is hiding information.
  + How
    - Declare fields/variables as private.
    - Provide public get and set methods, to access and update the value of a

private field.

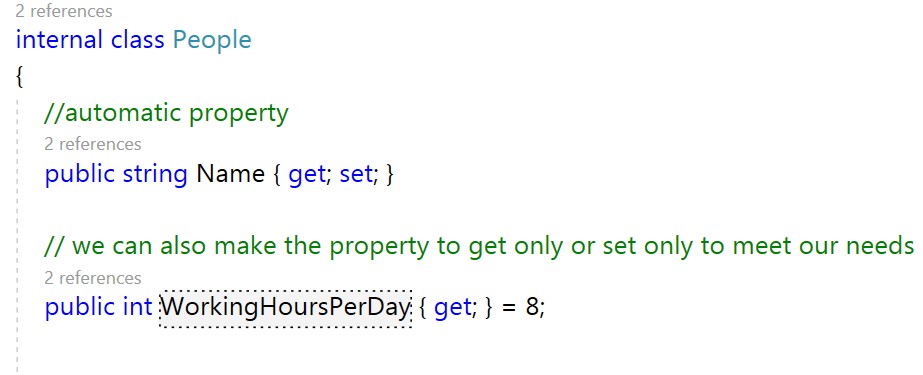
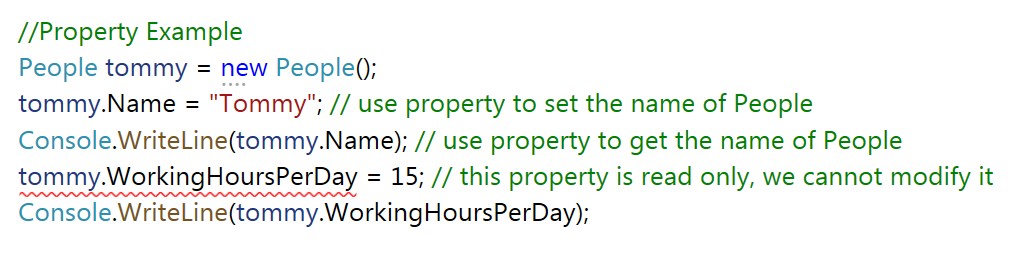
* + Why
    - Flexibility — Internal logic changes won’t affect the caller of the method
    - Reusability — Encapsulated code can be used by different callers
    - Maintainability — Operations on encapsulated unit won’t affect other parts

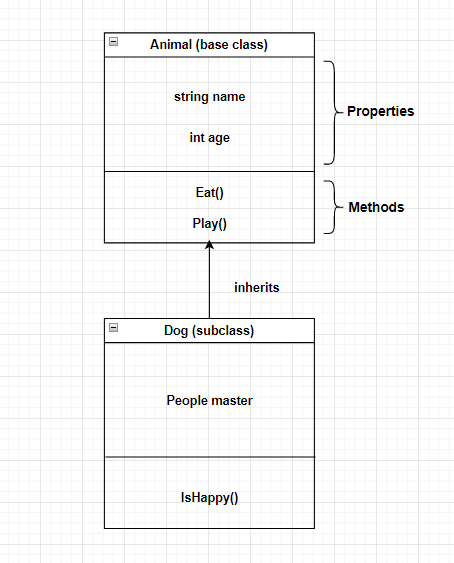
# Property



* + Property is an example of Encapsulation, we can use property to access and update the value of a private field.

# Automatic Property



Inheritance

* + Is the ability to derive something specific from something generic.
  + A class can inherit the features of another class and add its own modification.
  + The parent class is the base class and the child class is known as the subclass or derived class.
  + A subclass inherits all the properties and methods of the base class.
  + Aids in the reuse of code.

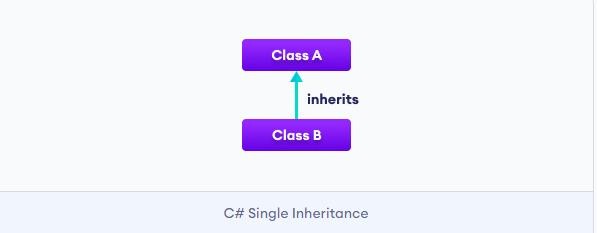
# Inheritance

* + Syntax
    - ***class* SubClass : BaseClass**
    - It declares that SubClass inherits the base class BaseClass
  + IS-A relationship
    - Inheritance implies that there is an IS-A relationship between subclass and base class
    - Eg. Dog is an Animal; Car is a Vehicle; Triangle is a Shape

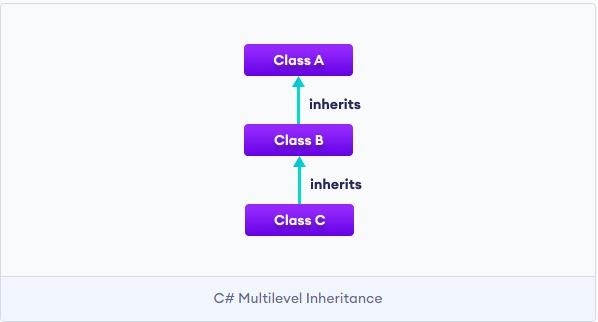
# Type of Inheritance

* + Single
  + Multiple
  + Multilevel
  + Hierarchical
  + Hybrid

# Single Inheritance



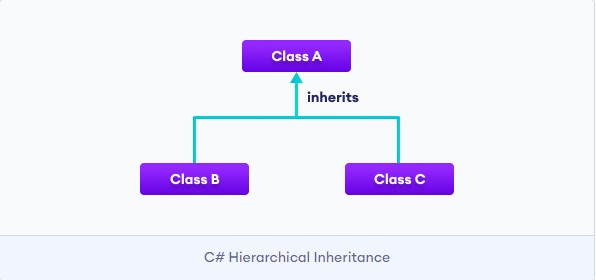
Multilevel Inheritance



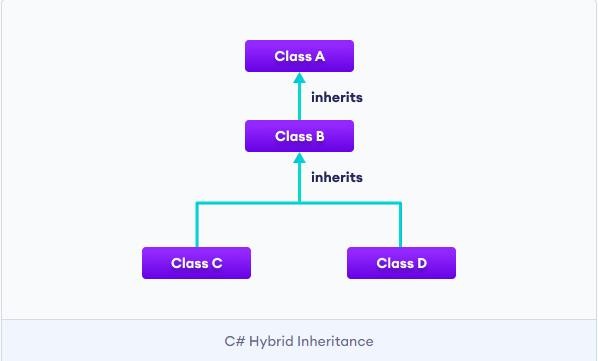
# Multiple Inheritance



Hierarchical Inheritance



# Hybrid Inheritance



Inheritance in C#

* + C# supports the following inheritance:
    - Single
    - Multi-level
    - Hierarchical
  + How about Multiple and Hybrid?

# Diamond Problem

* An ambiguity that can arise as a consequence of allowing multiple inheritance

# Is there any way to resolve the ambiguity caused by the diamond problem?

Interface

* + Like a class, an interface can have methods, but the methods declared in the interface are by default abstract (only method signature, no method body)
    - Interfaces specify what a class or a struct must do rather than how to

do.

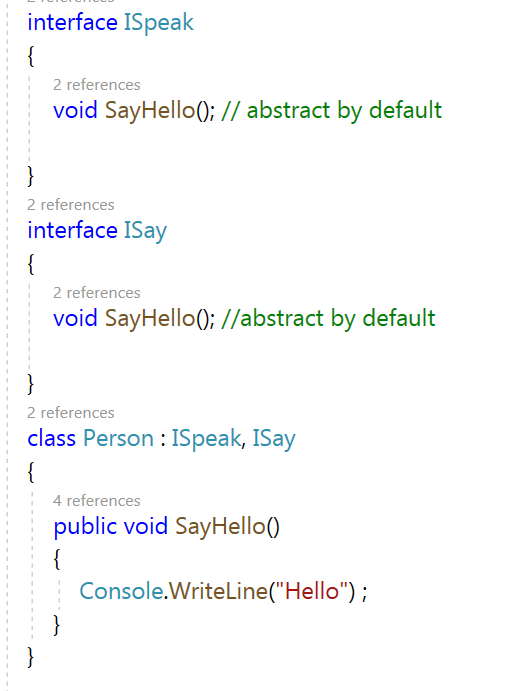
* + - It defines a contract, any class or struct that implements that contract must provide an implementation of the members defined in the interface.
    - If a class implements an interface and does not provide method

bodies for all functions specified in the interface, then the class must be declared abstract.

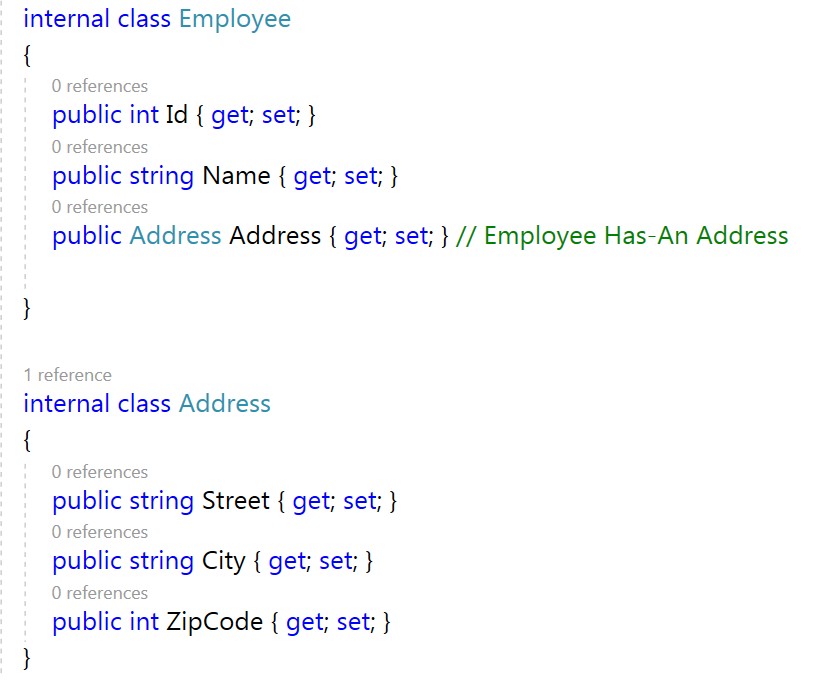
# Interface

* + Syntax
    - public *interface* IShape {}
    - public class Triangle **:** IShape {}
  + C# allows **multiple** inheritance of

interface



# Aggregation



* + If a class has an entity reference, it is known as Aggregation
  + Aggregation represents HAS-A relationship

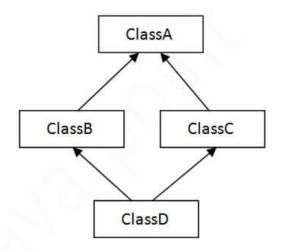
# Aggregation

* + Code reuse is also best achieved by aggregation when there is no

is-a relationship.

* + If we don’t have the Address class
    - Have to add all street, city information in employee
    - If we introduce an object called Company later which has the address too, we will have to add same attributes to Company object again

# Aggregation



* + How does Aggregation solve the diamond problem?
    - Instead of inheriting class B and C, introducing the B and C as the instance variables in class D
    - Thus, we can use b.DoSomething() or c.DoSomething() to eliminate the ambiguity

# Inheritance vs. Aggregation

* + Inheritance should be used only if the relationship Is-A is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

# Polymorphism

* + **Polymorphism** in c# is a concept by which we can perform

a single action in different ways.

* + - The word "poly" means many and "morphs" means forms. So

polymorphism means many forms.

* + There are two types of polymorphism in c#
    - **Compile time** polymorphism (achieved by **method overloading**)
    - **Runtime** polymorphism (achieved by **method overriding**)

# Compile time vs. Runtime

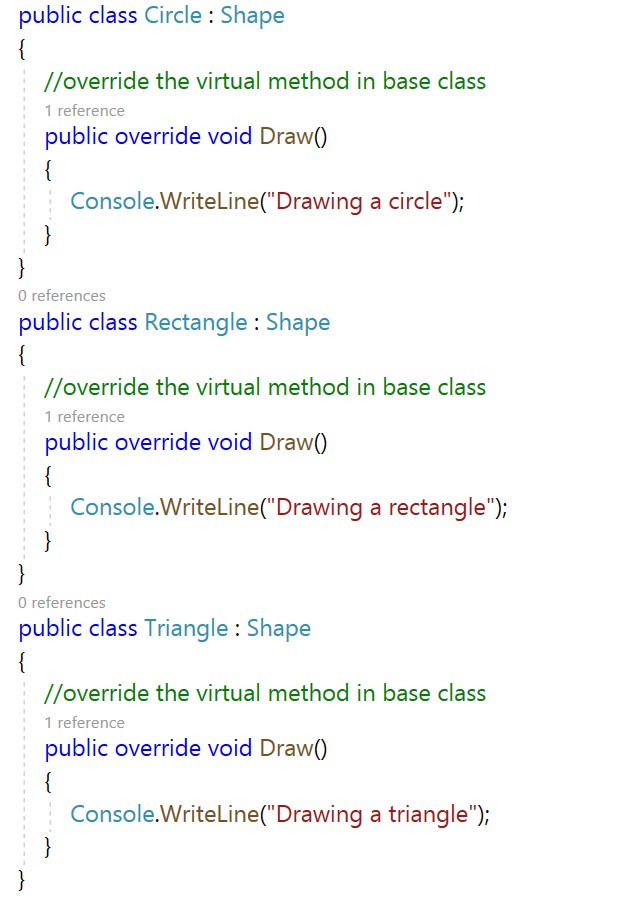
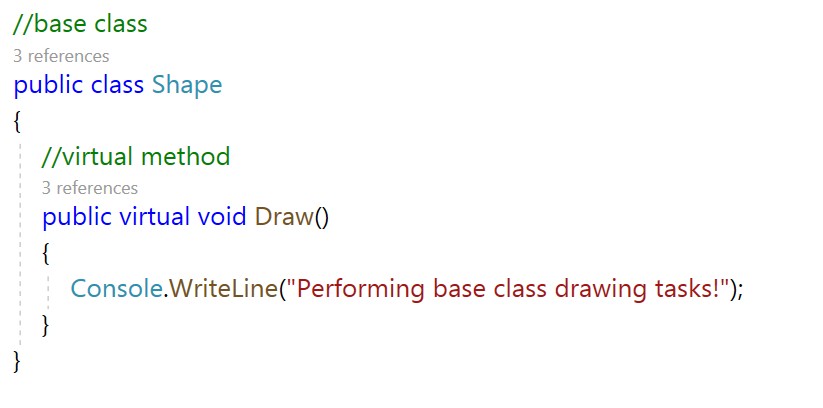
* + Compile time — the instance where the code you entered is converted to executable
  + Runtime — the instance where the executable is running

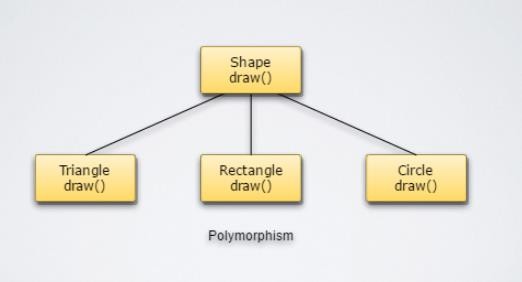
# Runtime Polymorphism

* + The form is determined at runtime
  + Method **overriding**
    - a method in a subclass has the **same name, return type, and parameters** as a method in its base class, then the method in the subclass is said to override the method in the base class
    - When an overridden method is called through the subclass object, it will always refer to the version of the method defined by the subclass. The base class version of the method is hidden.

# Virtual and Override Keywords

* + The ***virtual*** keyword is used to specify the virtual method in the base class, and the method with the same signature that needs to be overridden in the derived class is preceded by ***override*** keyword.
    - By default, methods are **non-virtual**. You cannot override a non-virtual method.
    - You cannot use the ***virtual*** keyword with the ***static***, ***private***, ***abstract***, or ***override*** keywords





# Compile Time Polymorphism

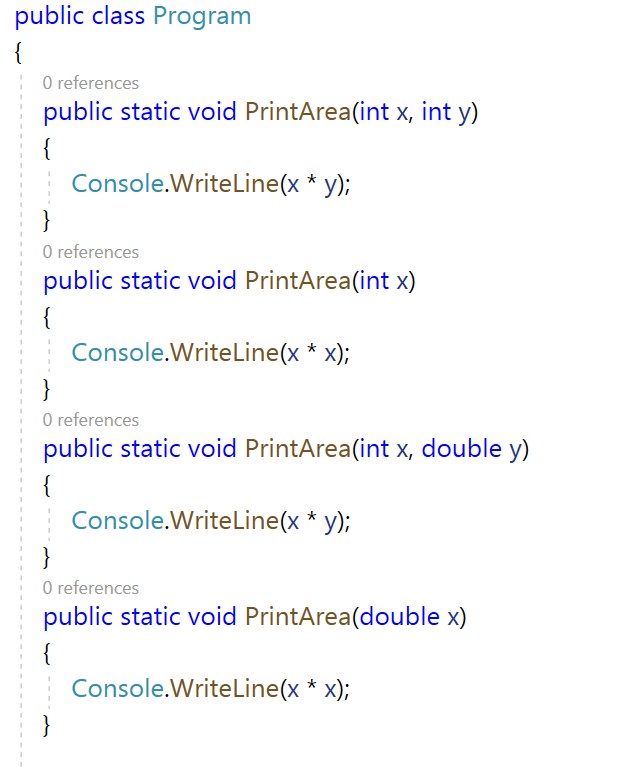
* + The form is determined at compile time
  + Method **overloading**
    - a class has multiple methods that have the same name but

different parameters

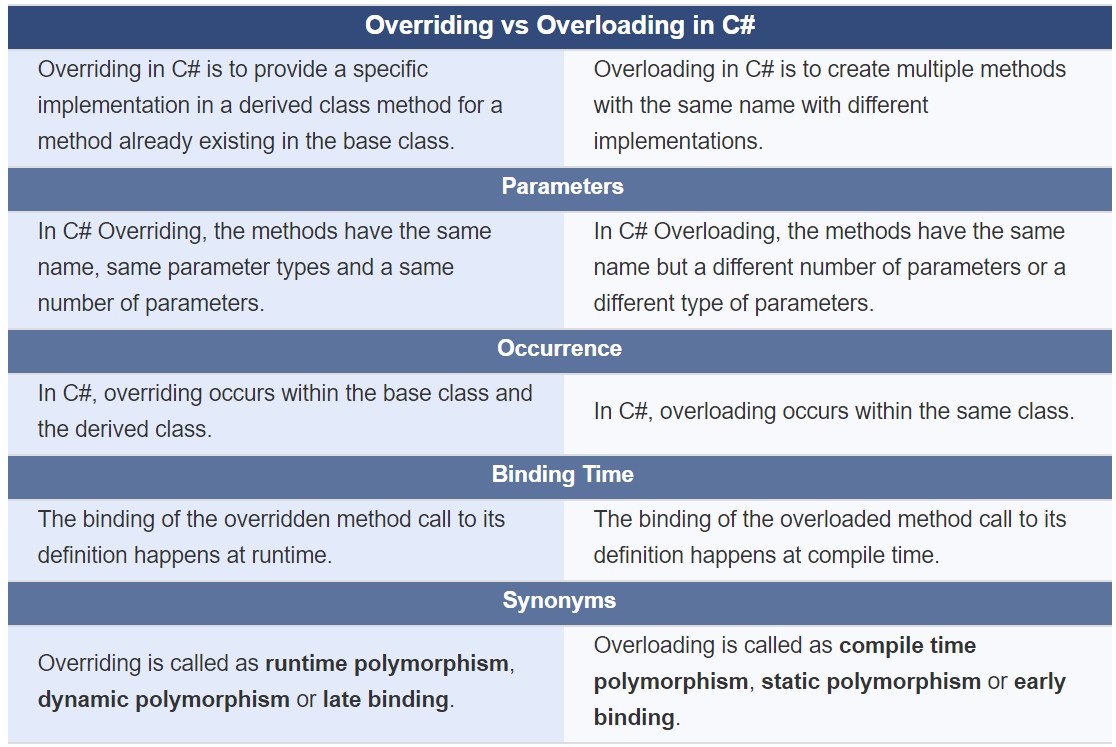
* + - * Different number of parameters
      * Different data types of parameters
    - Method overloading increases the readability of the

program

# Compile Time Polymorphism



Overriding vs. Overloading



# If I want to use the implementation in the base class, how to refer to the base class?

Base

* The **base** keyword is used to access members of the base class from

within a derived class

* Usage
  + Call a method on the base class that has been overridden by another

method.

* + Use base() to specify which base-class constructor should be called

when creating instances of the derived class.



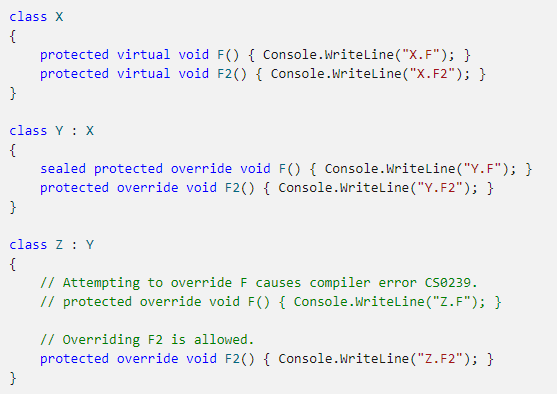
How can we prevent one method from being overridden?

# Sealed

●

Method – prevent overriding specific virtual methods

●



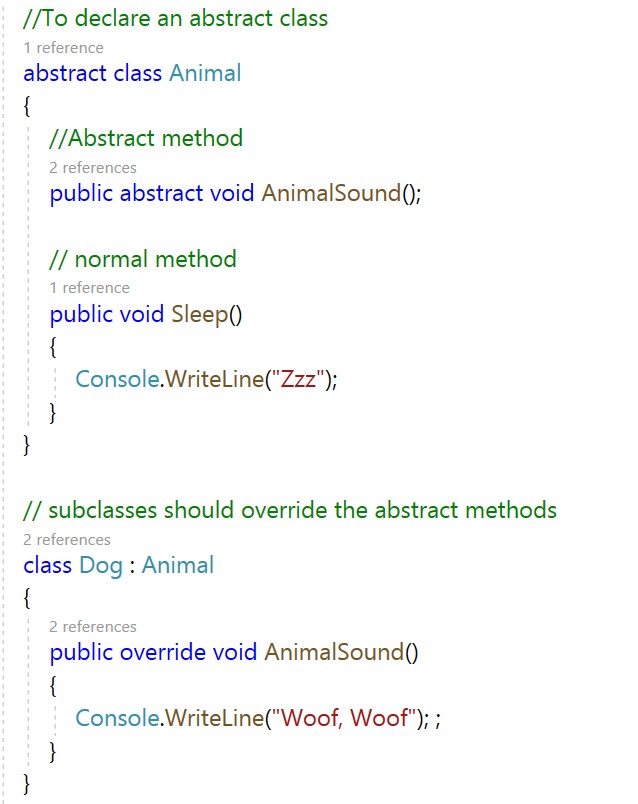
Class – prevent inheritance

* sealed class SealedClass {}

# Abstraction

* **Hiding internal details** and showing functionality
  + eg. phone call, we don't know the internal processing
* Abstraction is achieved by creating either **Abstract Classes** or **Interfaces** on top of your class
* Why
  + Hide the unnecessary things from user so providing easiness.
  + Hiding the internal implementation of software so providing

security



# Abstract Class

* Abstract classes are classes with a generic concept, not related to a specific class.
* Abstract classes define **partial behavior** and leave the rest for the subclasses to provide
* Contain zero or more abstract methods.
* Abstract method contains no implementation (like the method in Interface)
* Abstract classes **cannot be instantiated,** but they can have a reference variable
* If the subclasses do not override the abstract methods of the abstract class, then

it is mandatory for the subclasses to tag themselves as abstract.

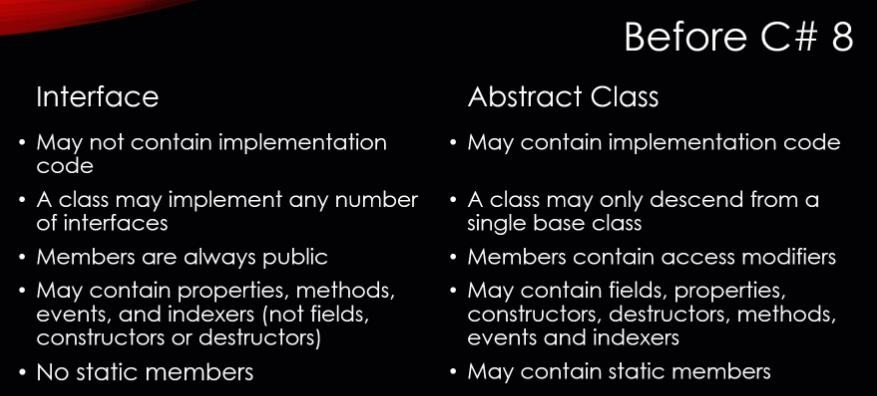
# Why Abstract Class

* To force the same name and signature pattern in all the subclasses
* To have the flexibility to code these methods with their own specific

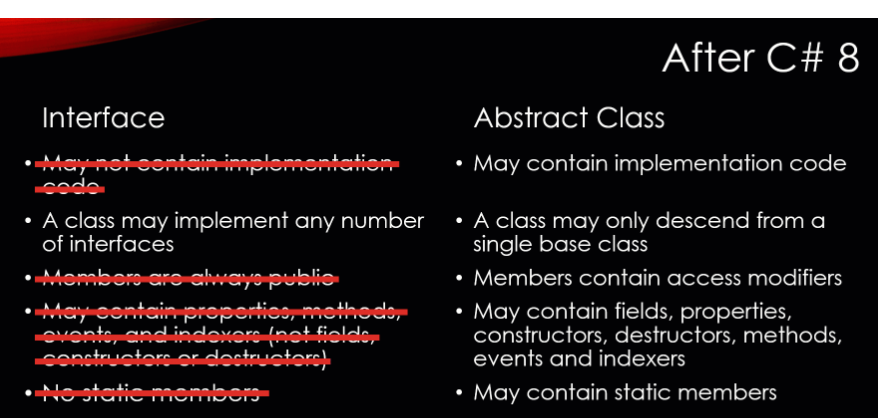
requirements

* To prevent accidental initialization
* To define common attributes or methods

# Abstract Class vs. Interface



Abstract Class vs. Interface



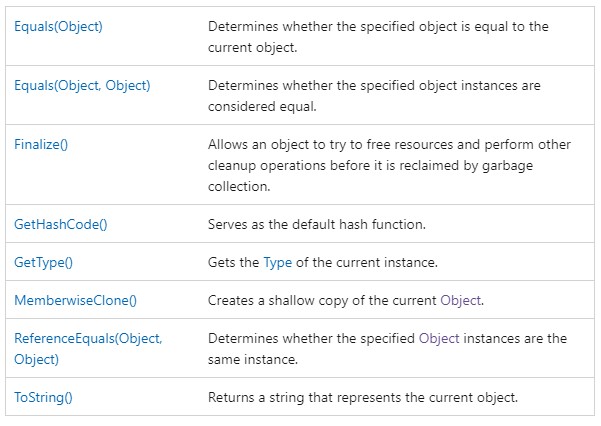
# Object Class

Object Class in C#

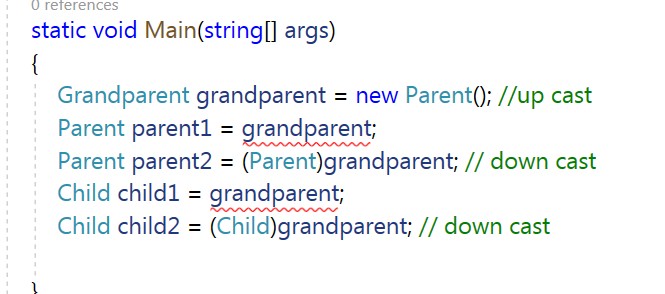
All types in the .NET type system **implicitly inherit** from [**Object**](https://docs.microsoft.com/en-us/dotnet/api/system.object) or a type derived from it. The common functionality of [**Object**](https://docs.microsoft.com/en-us/dotnet/api/system.object) is available to **any type**.

* The Object class is beneficial if you want to refer any object whose type you don't know. Notice that the parent class reference variable can refer to the child class object, known as **upcasting**

# Methods of Object Class



Object Casting



# Recap

* Class and Object
* Object-oriented programming (OOP)
* Encapsulation
  + Encapsulation
  + Inheritance
  + Polymorphism
  + Abstraction
* Object class

Question?

.NET Full Stack

Development Program

Day 4 Collections

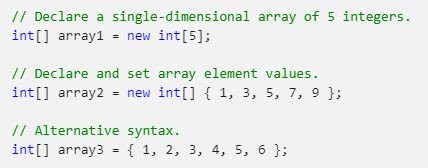
Outline

* Array
* Collections
  + Generic Collections
    - List, LinkedList, Dictionary, HashSet, SortedList, Stack, Queue
  + Non-generic Collections
    - ArrayList, HashTable, SortedList, Stack, Queue
* Comparisons and Sorts within Collections

# Collection

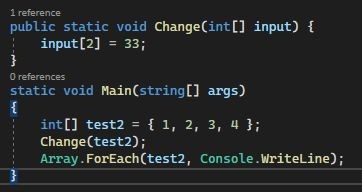
* Group of objects
* It is not speciﬁed whether they are
  + Ordered / not ordered
  + Duplicated / not duplicated
* Following constructors are common to all classes implementing Collection
  + T()
  + T(){...}
  + T(Collection c)

# Array

* Can store multiple variables of the **same type**
* Size of the array is **ﬁxed** when the array instance is created
* If you want the array to store elements of any type, you can specify **object** as its type
* The default values of numeric array elements are set to **zero**, and reference elements are set to **null**

# Array

* Passing Arrays as Argument



# Collections

* Generic Collections
  + Generic Collections work on the **speciﬁc type** that is speciﬁed in the program whereas non-generic collections work on the **object** type.
  + Using System.Collections.Generic;
  + List, LinkedList, Dictionary, HashSet, SortedList, Stack, Queue
* Non-generic Collections
  + In non-generic collections, each element can represent a value of a **different type**. The collection size is not ﬁxed. Items from the collection can be added or removed at runtime
  + Using System.Collections;
  + ArrayList, HashTable, SortedList, Stack, Queue

# Generic Collection

* List
  + List class is a collection that can be used for **speciﬁc types**.
  + List is a class that is similar to an array, but the size is not ﬁxed
  + Elements can be added / removed at runtime.
  + Ex. List<int> al = new List<int>();



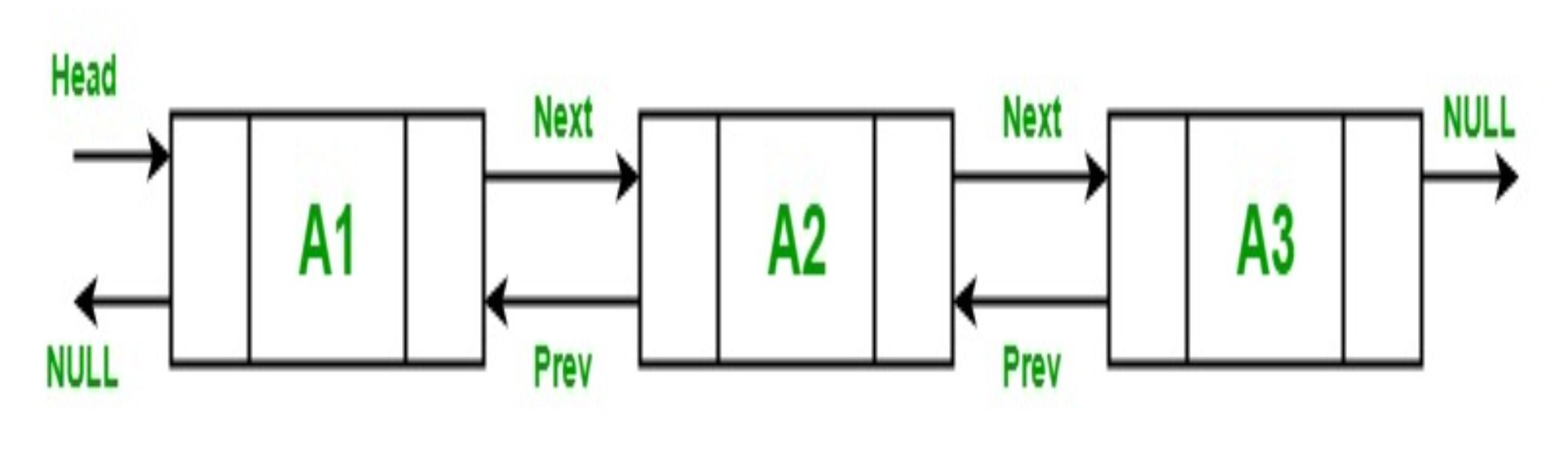
# Generic Collection

* List
  + access
  + Remove()
  + RemoveAt()
  + Contains()
  + IndexOf()
  + LastIndexOf()

# Generic Collection

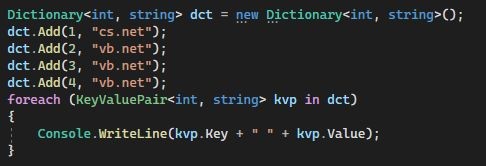
* LinkedList
  + a general-purpose linked list (doubly linked)
  + [LinkedList<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1?view=net-6.0) provides separate nodes of type [**LinkedListNode<T>**](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlistnode-1?view=net-6.0), so insertion and removal are **O(1)** operations.

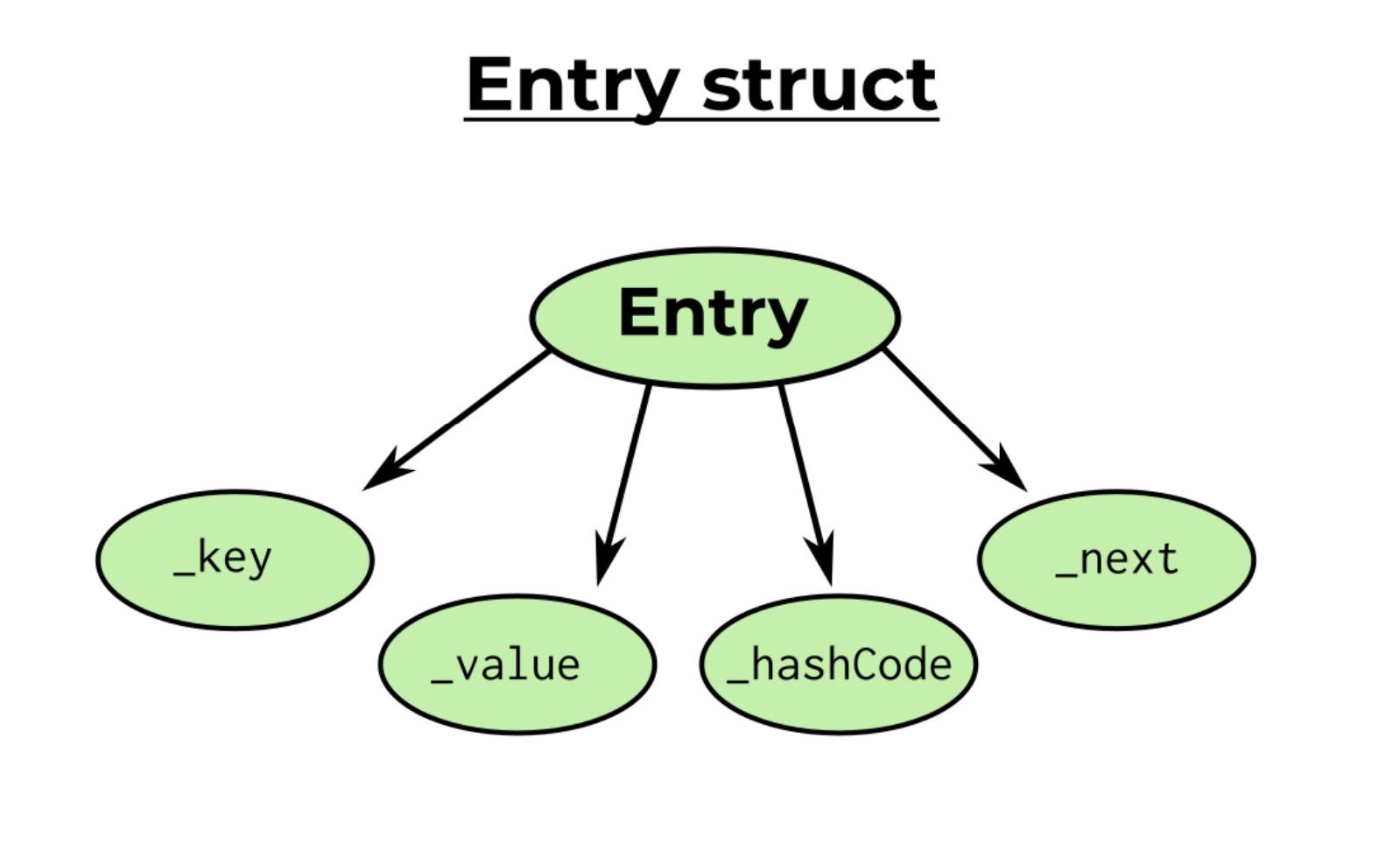


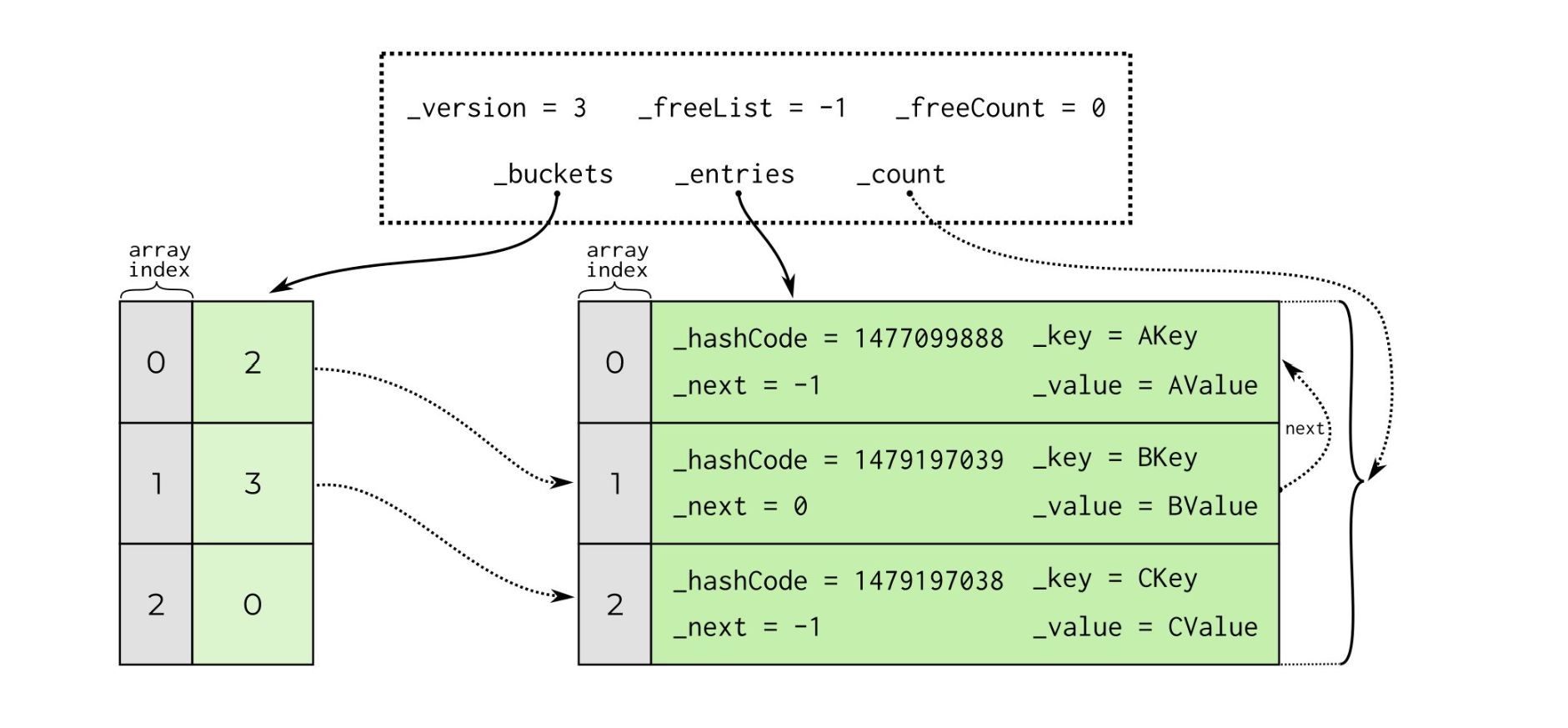


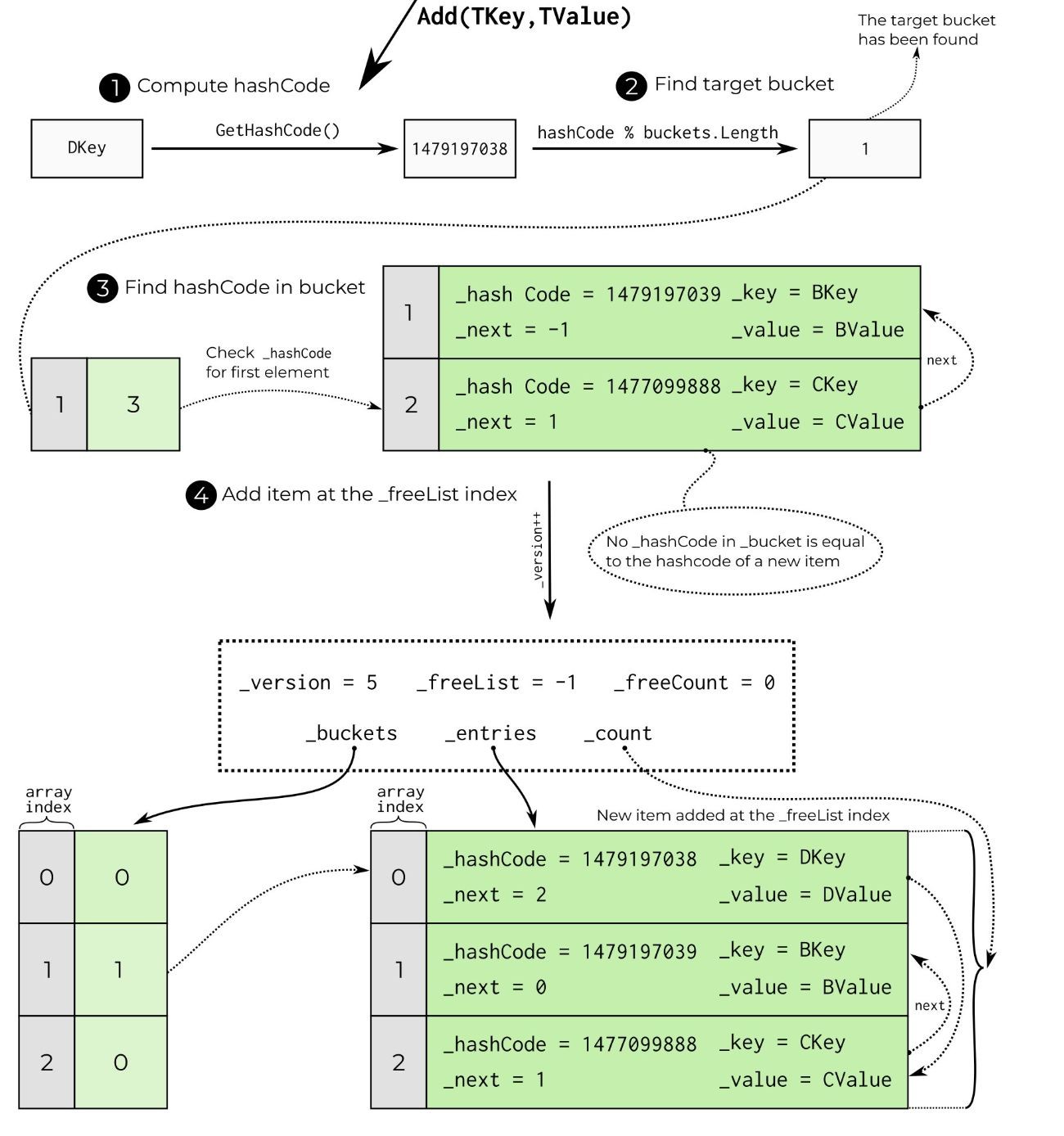
# Generic Collection

* Dictionary
  + represents the items as a combination of a key and value
  + access the value based on the key



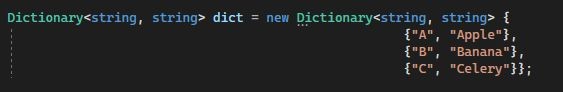






# Generic Collection

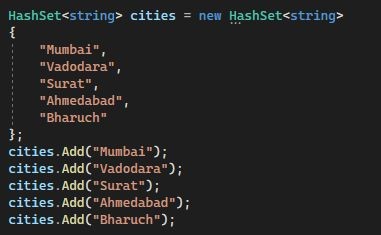
* Dictionary
  + Quick initialization



* + Methods
    - ContainsKey()
    - TryGetValue()
    - TryAdd()

# Generic Collection

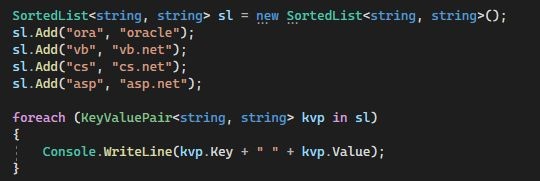
* HashSet
  + a collection that contains no duplicate elements, and whose elements are in no particular order



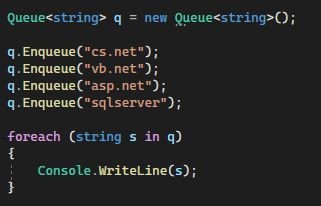
# Generic Collection

* SortedList
  + represents a collection of key/value pairs that are **sorted by key**

based on the associated [**IComparer<T>**](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.icomparer-1) implementation.



# Generic Collection

* Stack
  + It represents a last-in, ﬁrst out collection of object
* Queue
  + It represents a ﬁrst-in, ﬁrst out collection of object

# Non-generic Collection

* ArrayList
  + ArrayList class is a collection that can be used for **any types or objects**.
    - Arraylist is a class that is similar to an array, but it can be used to store values of various types.
    - An Arraylist doesn't have a speciﬁc size.
    - Any number of elements can be stored.
    - Ex. ArrayList al = new ArrayList();

# Non-generic Collection

* HashTable
  + represents the items as a combination of a key and value



# Non-generic Collection

* SortedList
  + is a class that has the combination of arraylist and hashtable.
  + represents a collection of key/value pairs that are **sorted by key** and are accessible by key and by index



# Non-generic Collection

* Stack
  + It represents a last-in, ﬁrst out collection of object
  + It is used when you need a last-in, ﬁrst-out access of items. When you add an item in the list, it is called pushing the item and when you remove it, it is called popping the item
* Queue
  + It represents a ﬁrst-in, ﬁrst out collection of object
  + It is used when you need a ﬁrst-in, ﬁrst-out access of items. When you add an item in the list, it is called enqueue and when you remove it, it is called deque

Comparison and Sorts within Collections

# Comparison (check for equality)

* If type T implements the [**IEquatable<T>**](https://docs.microsoft.com/en-us/dotnet/api/system.iequatable-1) generic interface, then the equality comparer is the [**Equals**](https://docs.microsoft.com/en-us/dotnet/api/system.iequatable-1.equals) method of that interface.
* If type T does **not** implement [**IEquatable<T>**](https://docs.microsoft.com/en-us/dotnet/api/system.iequatable-1), [**Object.Equals**](https://docs.microsoft.com/en-us/dotnet/api/system.object.equals) is used.

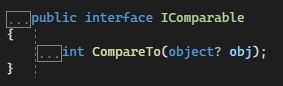
# Default Sorting

* Array.Sort(xxx) - using the System.IComparable
* xxx.Sort() – xxx is a collection – using default comparer (System.IComparable)
  + String objects are lexicographically ordered
  + Date objects are chronologically ordered
  + Number and sub-classes are ordered numerically

# Sort Order

* **IComparable<T>** interface
* **IComparer<T>** Interface

# IComparable<T> interface

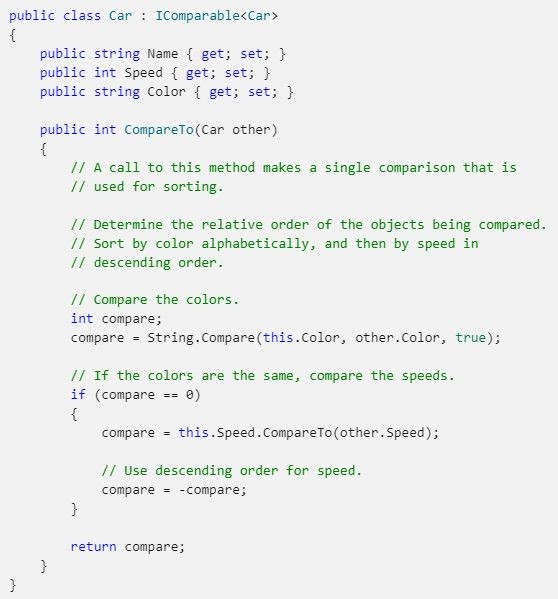


Compares the receiving object with the speciﬁed object

* Return value must be:
  + <0, if this precedes obj

○ ==0, if this has the same order as obj

* + >0, if this follows ob



# IComparer<T> Interface



Compares its two arguments

* Return value must be
  + <0, if x precedes y

○ ==0, if x has the same ordering as y

* + >0, if x follows y



Questions?

.NET Full Stack

Development Program

Day 5 I/O & Exception & C#Advance

Outline

* File Handling (System.IO)
* Exception
* Delegate
* Lambda Expression

File Handling

# System.IO

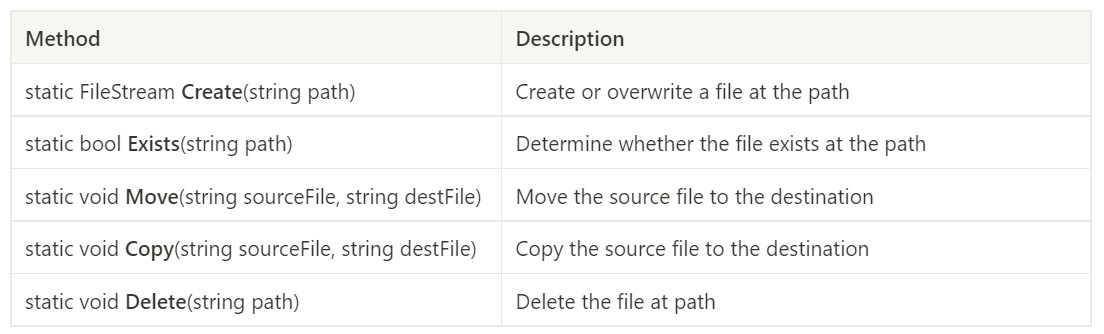
IO(Input and Output) is used to process the input and produce the output (read and write data). Most applications need to process some data and produce some output based on the input.

The **System.IO namespace** contains all the classes required for IO operations.

<https://learn.microsoft.com/en-us/dotnet/api/system.io?view=net-7.0>

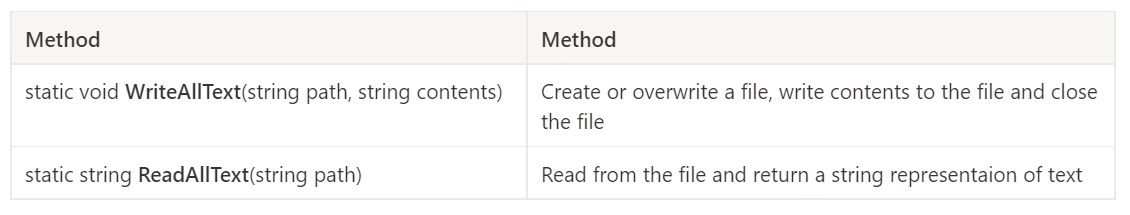
# File class

* C# include **static File class** to perform I/O operations on physical file.
* File class provides functionalities such as create, read/write, delete, etc. for physical files.
* Some methods in the File class:



# File class

* Read/Write methods in File class:

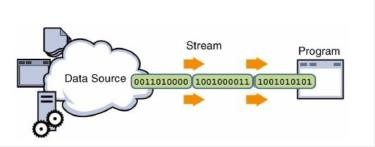
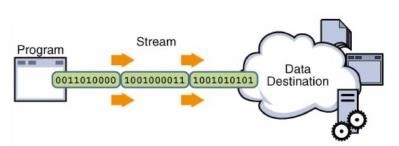


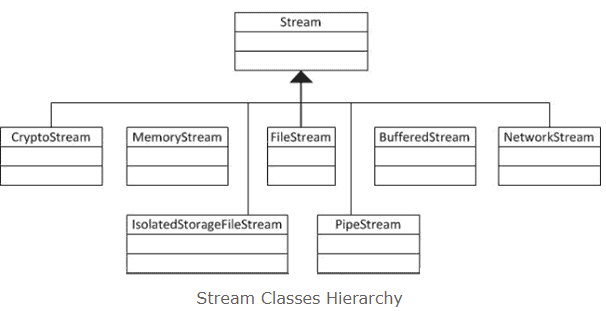
# File Class

* The static File class includes various utility methods to interact with a physical file
* It’s used to perform some quick operations on physical file
* It’s **NOT** recommended to use File class for multiple operations on multiple files at the same time due to performance reasons, a non-static class would be a better choice in this situation

# Stream

* C# uses the Stream to make IO operation fast.
* A stream is a conceptually endless flow of data. We can either read from a stream or write to a stream.
* A stream is connected to a data source or a data destination.
* It’s a good practice to close the stream after use.



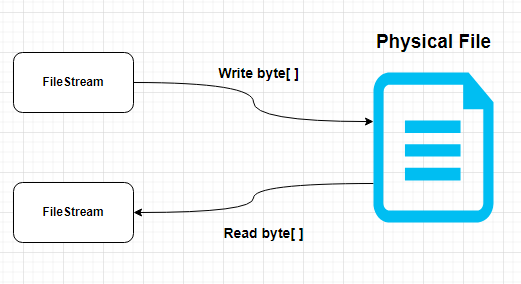


# Stream Class

* Stream is an abstract class that provides standard methods to transfer bytes to the source.
* The following classes inherit the Stream class to provide the functionality to Read/Write bytes from a particular source:

# FileStream Class

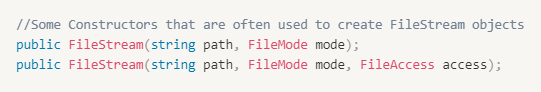
* FileStream class provides a stream for file operations. It can be used to perform

both **read** and **write operations**.

* To use the FileStream we need to:
  + **Include System.IO namespace**
  + **Create an instance of the FileStream**
  + **Perform read or write operations**
  + **Close the stream**

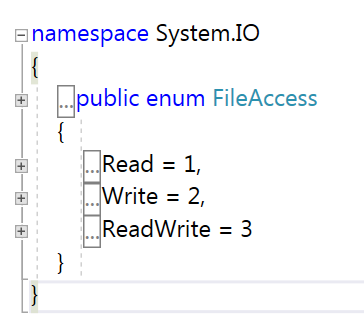
# Create an Instance of FileStream

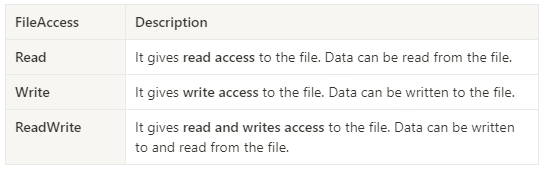
* We need an instance of the FileStream object to create new file or open an existing file.
* There are many constructors available, here are 2 simple ways:



* + **Path**: the path to the file that the current FileStream object will work on;
  + **FileMode**: specifies how to deal with the file(create, open, append …)
  + **FileAccess**: determines how the file can be accessed by the FileStream object(read, write…)

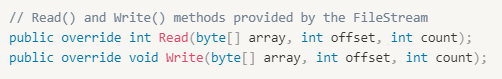
# FileMode

FileAccess



# FileStream

* Perform Read/Write operations:



* Close the Stream:
  + We can use the **Close()** method to close the stream(the connection to the file).

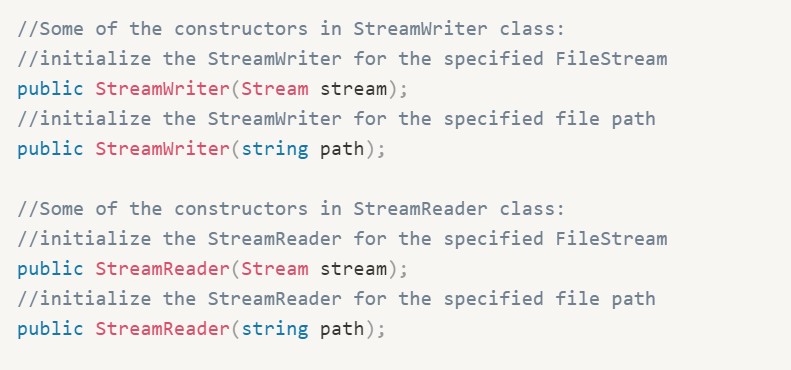
# StreamWriter and StreamReader

StreamWriter & StreamReader

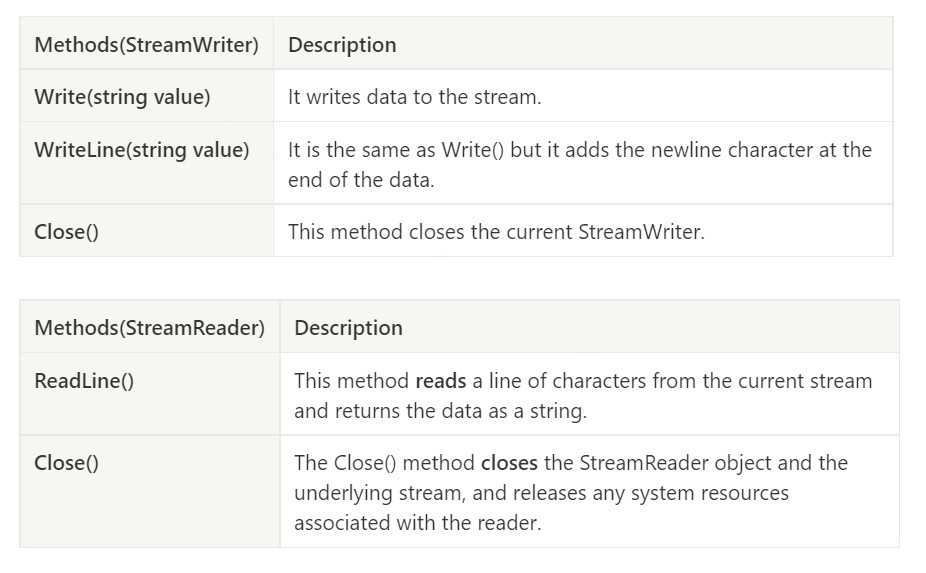
* StreamWriter: is a class for writing texts to a Stream by converting characters into bytes.
* StreamReader: is a class for reading characters from a Stream by converting bytes into characters.



# StreamWriter & StreamReader



StreamWriter & StreamReader



Serialization

# Serialization

* Serialization is **the process of converting an object into a stream of bytes** to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called **deserialization**.

Exception

# Exception

An **exception** is an unwanted or unexpected event, which occurs during the execution of a program

i.e at run time, that disrupts the normal flow of the program’s instructions. That means the statements placed after the exception-causing statements are not executed but the statements placed before that exception-causing statement are executed by CLR.

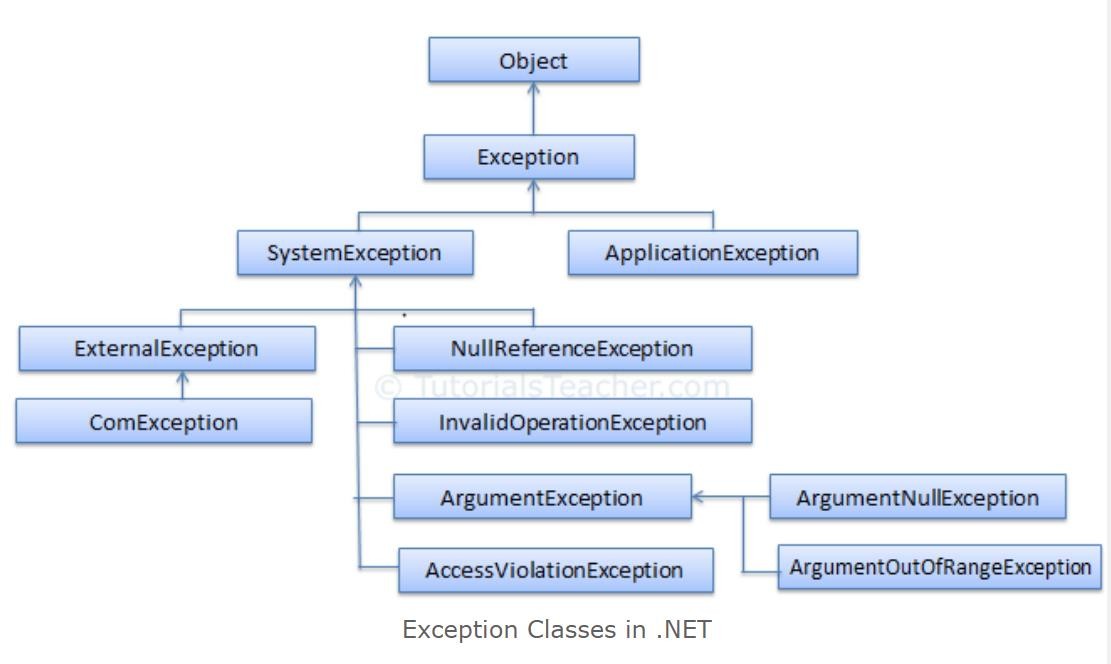
**Exception class** in C# is responsible for exceptions Exceptions can arise due to a number of situations.

* + Trying to access the 11th element of an array when the array contains only 10 elements.

(IndexOutOfRangeException)

* + Division by zero ([DivideByZeroException](https://docs.microsoft.com/en-us/dotnet/api/system.dividebyzeroexception))
  + Accessing a file which is not exist (FileNotFoundException)
  + Failure of I/O operations (IOException)

[Exception Hierarchy](https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/exceptions/compiler-generated-exceptions)



Exception Handling

* + try-catch
  + finally
  + throw

# try-catch

* + try/catch block can be placed within **any method** that you feel may

raise exceptions

* + All the statements to be tried for exceptions are put in a try block
  + catch block is used to catch any exception raised from the try block
  + If an exception occurs in any statement in the try block, the control immediately passes to the corresponding catch block

# try-catch

* + Order of catch is important
    - Always put smaller exceptions in front of bigger exceptions
    - Don’t put big basket in the front; you are going to catch

everything

* + - Don’t put small basket in the end; you will not catch anything
  + Nested try-catch block is allowed

# finally

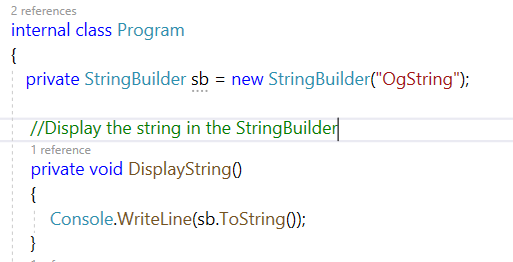
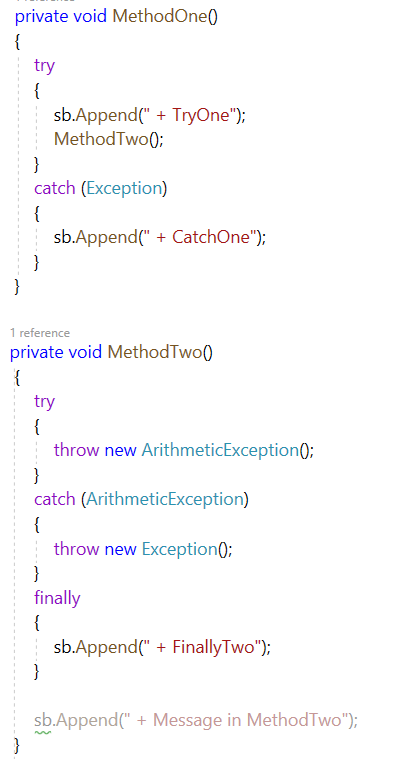
* + By using a ***finally*** block：

you can **clean up any resources** that are allocated in a try block you can **run code even if an exception occurs** in the try block.

* + Typically, ① the statements of a finally block run when control leaves a try

statement. ② The transfer of control can occur as a result of normal execution,

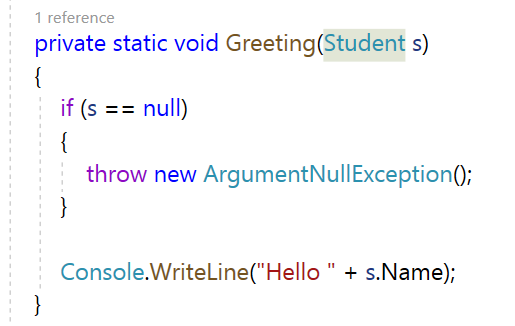
③ of execution of a return or break statement, ④ or of propagation of an exception out of the try statement.



# throw

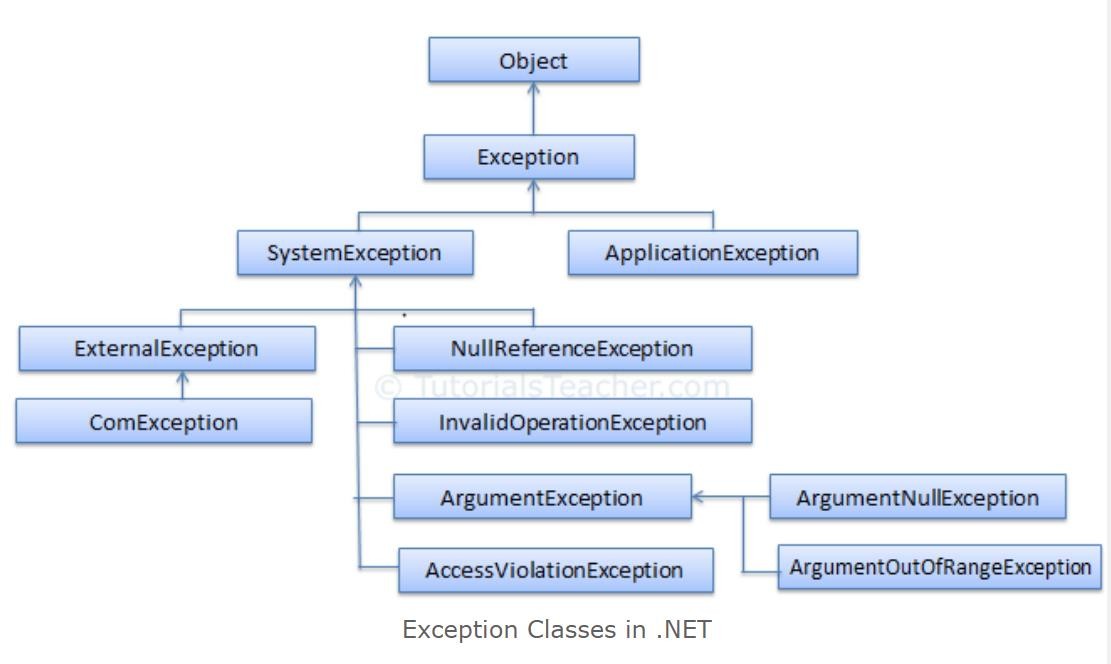
* + Used to explicitly throw an exception
  + Useful when we want to throw a user-defined exception.
  + The syntax for throw keyword is as follows:
    - throw new ThrowableInstance
      * eg. throw new ArgumentNullExcepton();

# throw



User-defined Exception

if none of the predefined exceptions meets your needs, you can create your own exception classes by deriving from the [**Exception**](https://docs.microsoft.com/en-us/dotnet/api/system.exception) class

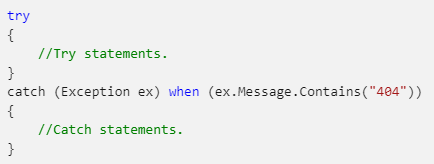


# User-defined Exception

* How to create your Exception:
  + Define a new class inheriting from the **Exception** class
  + Override the virtual members that are defined inside the Exception class

based on your need;

* + Throw the custom Exception instance where you need it



# User-filtered Exception Handlers

**User-filtered exception handlers** catch and handle exceptions based on requirements that you defined for the exception. This is useful when one catch statement(a particular exception object) corresponds to multiple exceptions;

use the ***catch*** statement with the ***when*** keyword, only when the condition in when(condition) is evaluated as true, the catch statement will get executed.

Delegate

# Delegate

* A delegate is a container for holding the reference of a method or function, it’s

described as a “Function Pointer”.

* A delegate can be declared using the ***delegate*** keyword, and it can be declared

within a class or a namespace(more common).

* The signature of the method must match the signature of the delegate.

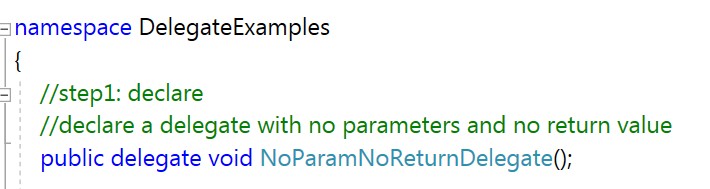


# Delegate

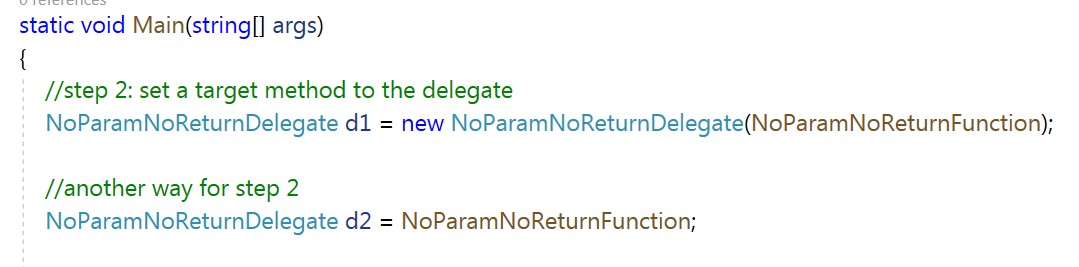
There are 3 steps involved while working with delegates:

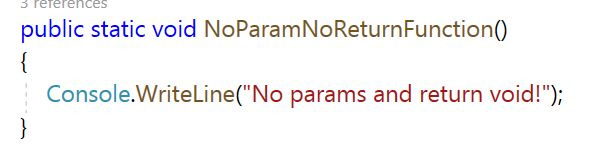
1. Declare a delegate (in a class or namespace)
2. Instantiate a delegate
3. Invoke a delegate

# Step1: Declare a Delegate

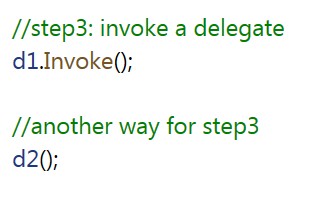
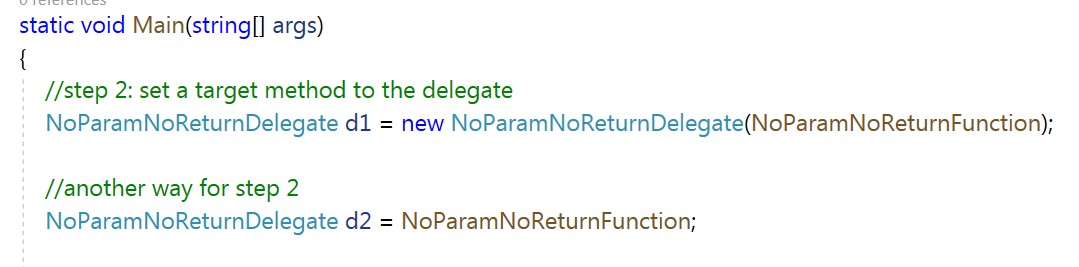


Step2: Instantiate a delegate





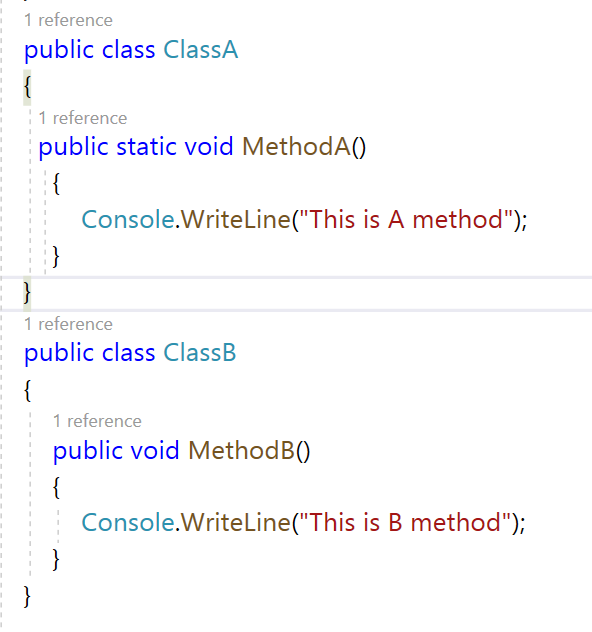
# Step3: Invoke a delegate



Multicast Delegate

* The delegate that points to multiple methods is called a multicast delegate.
* The addition operator adds a function to the invocation list, and the subtraction operator removes it.
* If a delegate returns a value, then the last assigned target method's value will

be return when a multicast delegate called.

# Delegate

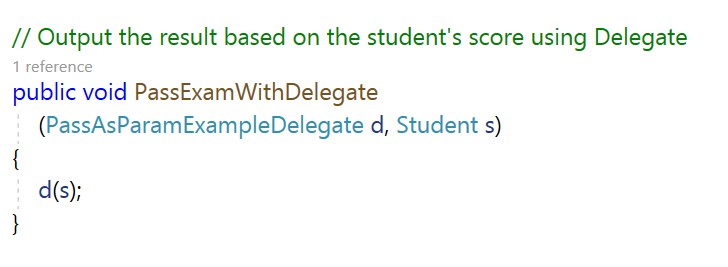
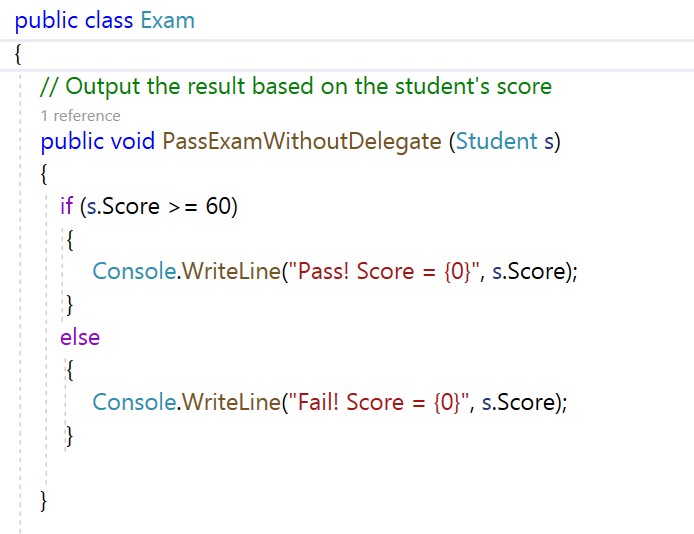
Delegates are often used as:

* + Members of a class
  + Parameters of a function/method

# Delegate

* (used as member of the class)





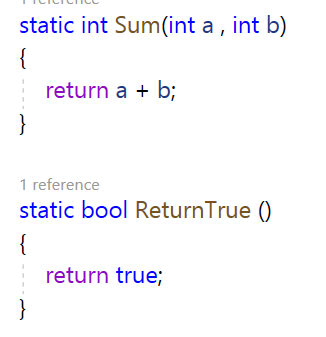
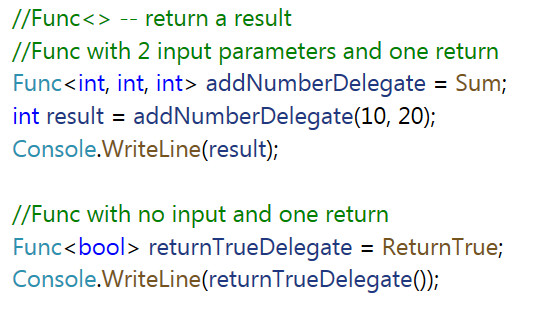
# Delegate

* (used as parameters of a function/method)

# Build-in Generic Delegates

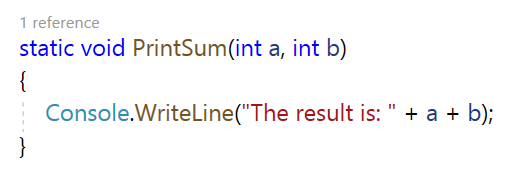
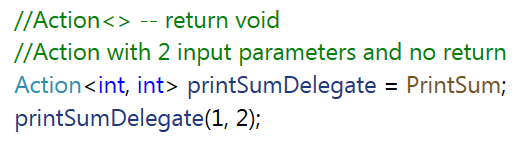
Func Delegate

* ***Func*** is a generic delegate included in the System namespace. It has zero or more input parameters and one out parameter. The last parameter is considered as an out parameter.
* Func<params\_types..., return\_type> variableName;



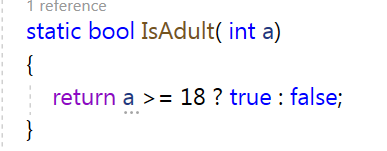
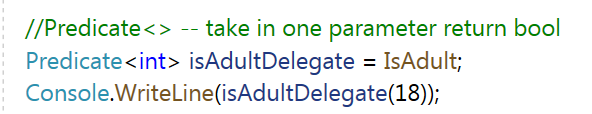
# Action Delegate

* ***Action*** is a delegate type defined in the System namespace. An Action type delegate is the same as ***Func*** delegate except that the Action delegate doesn't return a value. In other words, an Action delegate can be used with a method that has a void return type.
* Action<params\_types, ...> variableName;

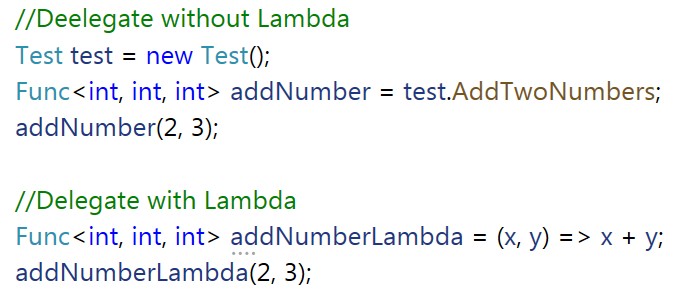


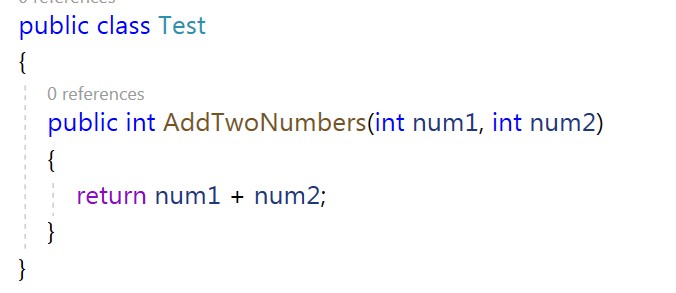
# Predicate Delegate

* A ***predicate*** delegate method must take one input parameter and return bool - true or false.



# Lambda Expressions

* Lambda expression in C# is the shorthand for representing anonymous method.
* Syntax:
  + Parameters => Body expression



# Lambda Expression

* Lambda expression can have zero parameter:
* Lambda expression can have multiple parameters in parenthesis ():
* Lambda Expression can have multiple statements in body expression in curly brackets {}
* Lambda Expression can be assigned to Func, Action or Predicate delegate



# Outline

* File Handling (System.IO)
* Exception
* Delegate
* Lambda Expression

# Question?